bdiAccess

Interface Specification

Version Control:

Version	Date	Remark
1.00	13.07.98	First version, based on document bdiifc.doc
1.04	27.10.98	BDI_UpdateFirmwareLogic(), BDI_InfoGet() added
1.05	12.03.99	Support for HC12 targets added
1.06	29.04.99	Support for ARM targets added BDI_ChannelRead(), BDI_ChannelWrite() added
1.08	23.12.99	Support for COP targets added (PPC6xx/7xx/82xx)
1.09	27.12.99	Support for MPC555 flash programming added
1.10	06.03.00	Support for BDI1000 added, Parameter changed for ARM in BDI_TargetStartup()
1.11	12.05.00	Support for PPC400 added
1.12	08.06.00	Redefinition of BDI_TargetStartup() parameter for PPC6xx/7xx/82xx targets. Additional init list commands for PPC6xx/7xx/82xx targets.
1.13	14.07.00	Support for Linux/Unix hosts added.
1.14	05.09.00	Support for new HC12 chips added.
1.15	14.09.00	Support for Atmel AT49 / SST flash added. Additional flash erase function with erase mode as parameter added (BDI_FlashErase()).
1.16	03.10.00	Support for MMC2107 added.
1.17	25.10.00	Improved flash programming for MPC555 internal flash. Parameter changed for COP in BDI_TargetStartup().
1.18	09.01.01	Support for MPC7400 added
1.19	30.04.01	Firmware update for BDI1000 and BDI2000 Rev.C added.
1.22	07.12.01	Functions to setup HC12 automatic programming mode added. More information about how to program HC12 flash added.
1.23	04.03.02	Network configuration function added. Support for MIPS32 added.
1.24	24.04.02	Support for XScale added. Description of CP register access added. Description how to unsecure newer S12 devices added. BDI_GetConfiguation() function added.
1.25	23.07.02	Support for MPC565 internal flash added. Info about PPC440 TLB entries added.
1.26	06.08.02	Support for AMD MirrorBit flash added.
1.27	30.10.02	Additional information for TI925T added
1.28	xx.01.03	Support for paged addresses in BDI_EnableAutoProgHC12(). Support for HCS12 with 512k flash added. Support for 32 bit wide flash added.
1.29	02.05.03	Some additional informations added.
1.30	30.06.03	Support for Coldfire Flash Module (CFM) added.
1.31	08.07.03	Support for MMC2114 Flash Module (SGFM) added.
1.32	05.09.03	Some errors in this document corrected.
1.33	16.10.03	Support for MAC7100 internal flash programming added.

Version	Date	Remark
1.34	05.03.04	Some minor extension.
1.35	05.07.04	Support for MPC85xx added.
1.36	15.07.04	Support for MC9S12X processor family added
1.37	31.08.04	Support for ARM11 added. Progess callback for file write functions added. Dump, Load, Verify and Program for binary files added.
1.38	08.10.04	Support for internal flash of LPC2000, ADuC7000 and STA2051 added.
1.39	01.11.04	Support for ST30F7xx internal flash added.
1.40	20.04.05	Support for internla flash of Atmel AT91SAM7S added. Support for Freescale MPC83xx (e300) added. Progress callback can abort current file write function.
1.41	24.02.06	Make document version identical to library version.
1.42	25.08.06	Support for STR91xF internal flash added.
1.43	07.02.07	New BDI_SetOption() function added.
1.44	09.08.07	Support for MPC744x/745x added
1.45	17.08.07	Low level JTAG access added. BDI3000 specific information added.
1.46	13.12.07	Function BDI_ResetInitList() and BDI_AddInitList() added.

©ABATRON AG

Currently Available Functions:

	CPU12	CPU16	CPU32	MCF	M-Core	MPC5xx/8xx	ARM
BDI_Connect	X	X	X	X	X	X	X
BDI_Disconnect	X	X	X	X	X	X	X
BDI_InstallCallback	X	X	X	X	X	X	X
BDI_FileSize	X	X	X	X	X	X	X
BDI_RegisterGet	X		X	X	X	X	X
BDI_RegisterSet	X		X	X	X	X	X
BDI_FPRegGet						X	
BDI_FPRegSet						X	
BDI_VRegGet							
BDI_VRegSet							
BDI_GetByte	X	X	X	X	X	X	X
BDI_GetWord	X	X	X	X	X	X	X
BDI_GetLong	X	X	X	X	X	X	X
BDI_SetByte	X	X	X	X	X	X	X
BDI_SetWord	X	X	X	X	X	X	X
BDI_SetLong	X	X	X	X	X	X	X
BDI_DumpBlock	X	X	X	X	X	X	X
BDI_DumpFile	X	X	X	X	X	X	X
BDI_DumpBinary	X	X	X	X	X	X	X
BDI_LoadBlock	X	X	X	X	X	X	X
BDI_LoadFile	X	X	X	X	X	X	X
BDI_LoadBinary		X	X	X	X	X	X
BDI_VerifyBlock	X	X	X	X	X	X	X
BDI_VerifyFile	X	X	X	X	X	X	X
BDI_VerifyBinary		X	X	X	X	X	X
BDI_TargetReset	X	X	X	X	X	X	X
BDI_InitListReset	X			X	X	X	X
BDI_InitListAdd	X			X	X	X	X
BDI_TargetStartup	X			X	X	X	X
BDI_SetBdmSpeed	X	X	X	X		X	
BDI_SetOption							X
BDI_TargetStart	X	X	X	X	X	X	X
BDI_TargetHalt	X	X	X	X	X	X	X
BDI_StateGet	X	X	X	X	X	X	X
BDI_ErrorTextGet	X	X	X	X	X	X	X
BDI_ChannelRead							X
BDI_ChannelWrite							X
BDI_FlashSetType			X	X	X	X	X
BDI_FlashEraseSector			X	X	X	X	X
BDI_FlashErase			X	X	X	X	X
BDI_FlashWriteBlock	X		X	X	X	X	X
BDI_FlashWriteFile	X		X	X	X	X	X
BDI_FlashWriteBinary			X	X	X	X	X
BDI_FlashSetupHC12	X						
BDI_FlashEraseHC12	X						
BDI_EnableAutoProgHC12	X						
BDI_DisableAutoProgHC12	X						
BDI_FlashSetup555						X	
BDI_FlashErase555						X	
BDI_FlashWrite555						X	
BDI_FlashSetupUC3F						X	
BDI_FlashEraseUC3F						X	
BDI_FlashSetupLPC2000							X
BDI_InfoGet	X	X	X	X	X	X	X
BDI_UpdateFirmwareLogic	X	X	X	X	X	X	X
BDI_ConfigNetwork	X	X	X	X	X	X	X
BDI_GetConfiguation	X	X	X	X	X	X	X
BDI_DoJtag							X

	MIPS32	6xx/7xx	PQ3	PPC4xx	MPC5500	XScale	
BDI_Connect	X	X	X	X	X	X	
BDI_Disconnect	X	X	X	X	X	X	
BDI_InstallCallback	X	X	X	X	X	X	
BDI_FileSize	X	X	X	X	X	X	
BDI_RegisterGet	X	X	X	X	X	X	
BDI_RegisterSet	X	X	X	X	X	X	
BDI_FPRegGet		X	X	X	X		
BDI_FPRegSet		X	X	X	X		
BDI_VRegGet		X					
BDI_VRegSet		X					
BDI_GetByte	X	X	X	X	X	X	
BDI_GetWord	X	X	X	X	X	X	
BDI_GetLong	X	X	X	X	X	X	
BDI_SetByte	X	X	X	X	X	X	
BDI_SetWord	X	X	X	X	X	X	
BDI_SetLong	X	X	X	X	X	X	
BDI_DumpBlock	X	X	X	X	X	X	
BDI_DumpFile	X	X	X	X	X	X	
BDI_DumpBinary	X	X	X	X	X	X	
BDI_LoadBlock	X	X	X	X	X	X	
BDI_LoadFile	X	X	X	X	X	X	
BDI_LoadBinary	X	X	X	X	X	X	
BDI_VerifyBlock	X	X	X	X	X	X	
BDI_VerifyFile	X	X	X	X	X	X	
BDI_VerifyBinary	X	X	X	X	X	X	
BDI_TargetReset	X	X	X	X	X	X	
BDI_ InitListReset	X	X	X	X	X	X	
BDI_ InitListAdd	X	X	X	X	X	X	
BDI_TargetStartup	X	X	X	X	X	X	
BDI_SetBdmSpeed							
BDI_ SetOption							
BDI_TargetStart	X	X	X	X	X	X	
BDI_TargetHalt	X	X	X	X	X	X	
BDI_StateGet	X	X	X	X	X	X	
BDI_ErrorTextGet	X	X	X	X	X	X	
BDI_ChannelRead							
BDI_ChannelWrite							
BDI_FlashSetType	X	X	X	X	X	X	
BDI_FlashEraseSector	X	X	X	X	X	X	
BDI_FlashErase	X	X	X	X	X	X	
BDI_FlashWriteBlock	X	X	X	X	X	X	
BDI_FlashWriteFile	X	X	X	X	X	X	
BDI_FlashWriteBinary	X	X	X	X	X	X	
BDI_FlashSetupHC12							
BDI_FlashEraseHC12							
BDI_EnableAutoProgHC12							
BDI_DisableAutoProgHC12							
BDI_FlashSetup555							
BDI_FlashErase555							
BDI_FlashWrite555							
BDI_FlashSetupUC3F							
BDI_FlashEraseUC3F							
BDI_FlashSetupLPC2000							
BDI_InfoGet	X	X	X	X	X	X	
BDI_UpdateFirmwareLogic	X	X	X	X	X	X	
BDI_ConfigNetwork	X	X	X	X	X	X	
BDI_GetConfiguation	X	X	X	X	X	X	
BDI_DoJtag				X			

Table of contents

1. Overview 1
1.1. BDI-HS (old product, no longer produced) 1
1.2. BDI2000 / BDI1000 1
1.3. Windows Hosts 2
1.4. Linux/ Unix Hosts 2
2. BDI Access Library Functions 3
2.1. Communication Functions 3
2.1.1. BDI_Connect() 3
2.1.2. BDI_Disconnect() 4
2.2. Register Access Functions 5
2.2.1. BDI_RegisterGet() 5
2.2.2. BDI_RegisterSet() 5
2.2.3. BDI_FPRegGet() 12
2.2.4. BDI_FPRegSet() 12
2.2.5. BDI_VRegGet() 13
2.2.6. BDI_VRegSet() 13
2.3. Memory Access Functions 14
2.3.1. BDI_GetByte(), BDI_GetWord(), BDI_GetLong() 14
2.3.2. BDI_SetByte(), BDI_SetWord(), BDI_SetLong() 14
2.3.3. BDI_DumpBlock() 15
2.3.4. BDI_DumpFile() 15
2.3.5. BDI_DumpBinary() 15
2.3.6. BDI_LoadBlock() 16
2.3.7. BDI_LoadFile(), BDI_LoadFile2() 16
2.3.8. BDI_LoadBinary() 16
2.3.9. BDI_VerifyBlock() 17
2.3.10. BDI_VerifyFile(), BDI_VerifyFile2() 17
2.3.11. BDI_VerifyBinary() 17
2.4. Target Control Functions 18
2.4.1. BDI_TargetReset() 18
2.4.2. BDI_SetBdmSpeed() 19
2.4.3. BDI_SetOption() 20
2.4.4. BDI_TargetStartup() 21
2.4.5. BDI_TargetStart() 42
2.4.6. BDI_TargetHalt() 42
2.4.7. BDI_StateGet() 43

2.5. Flash Programming Functions ... 51
2.5.1. BDI_ FlashSetType() ... 51
2.5.2. BDI_ FlashEraseSector() ... 54
2.5.3. BDI_ FlashErase() ... 54
2.5.4. BDI_ FlashWriteBlock() ... 55
2.5.5. BDI_ FlashWriteFile(), BDI_ FlashWriteFile2() ... 55
2.5.6. BDI_ FlashWriteBinary() ... 55
2.5.7. BDI_FlashSetupHC12() ... 56
2.5.8. BDI_FlashEraseHC12() ... 57
2.5.9. BDI_EnableAutoProgHC12() ... 58
2.5.10. BDI_DisableAutoProgHC12() ... 60
2.5.11. BDI_FlashSetup555() ... 61
2.5.12. BDI_FlashErase555() ... 62
2.5.13. BDI_FlashWrite555() ... 63
2.5.14. BDI_FlashSetupUC3F() ... 64
2.5.15. BDI_FlashEraseUC3F() ... 65
2.5.16. BDI_FlashSetupSGFM() ... 66
2.5.17. BDI_FlashSetupLPC2000() ... 67

2.6. Miscellaneous Functions ... 68
2.6.1. BDI_InfoGet() ... 68
2.6.2. BDI_UpdateFirmwareLogic() ... 71
2.6.3. BDI_ConfigNetwork() ... 72
2.6.4. BDI_GetConfiguation () ... 73
2.6.5. BDI_ErrorTextGet() ... 74
2.6.6. BDI_ChannelRead(), BDI_ChannelWrite() ... 75
2.6.7. BDI_InstallCallback(), BDI_FileSize() ... 76
2.6.8. BDI_DoJtag() ... 77

2.7. Additional information ... 80
2.7.1. Flash Programming for HC12 ... 80
2.7.1.1. Internal EEPROM ... 80
2.7.1.2. Internal Flash ... 82
2.7.1.3. External Flash ... 86
2.7.1.4. Working in debug mode ... 88
2.7.1.5. Unsecuring a HCS12 device ... 89
2.7.2. Programming the ColdFire Flash Module (CFM) ... 90
2.7.3. Programming the MAC7100 Flash Module (CFM32, CFM16) ... 91
2.7.4. Programming the ADuC7020 Flash Memory ... 92
2.7.5. Programming the ST STA2051 Flash Memory ... 93
2.7.6. Programming the ST ST30F774 Flash Memory ... 94
2.7.7. Programming the Atmel AT91SAM7Sxx Flash Memory ... 95

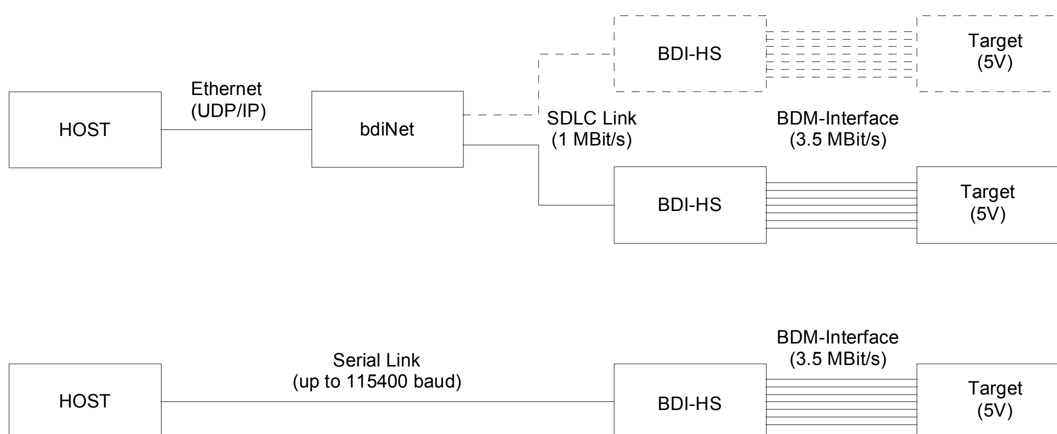
1. Overview

The bdiAccess library can be used to access the BDI functions. With this library, a user can execute the following task:

- Read/Write target registers
- Read/Write target memory
- Reset/Startup target
- Flash programming
- Start/Stop program execution

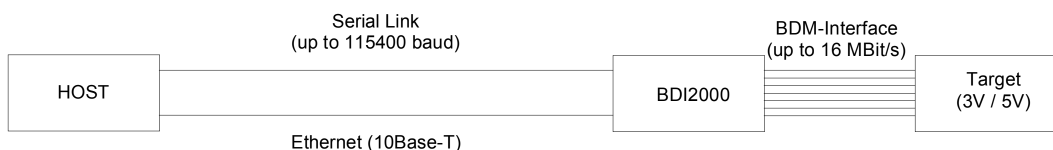
1.1. BDI-HS (old product, no longer produced)

The BDI-HS can connect directly to the host using a serial port or can connect to the bdiNet (the Ethernet Interface). The Ethernet link uses UDP/IP. The two BDI-HS that can connect to one bdiNet are addressed with two different UDP-Port numbers.



1.2. BDI3000 / BDI2000 / BDI1000 (BDI)

The BDI can connect directly to the host using a serial port or Ethernet.



1.3. Windows Hosts

For Windows hosts, the bdiAccess functions are accessed using a DLL. The name of the library is BDIIFC32.DLL for the 32bit version and BDIIFC16.DLL for the 16bit version of Windows.
The 16bit version is no longer supported.

1.4. Linux/ Unix Hosts

For Linux/Unix Hosts the bdiAccess library is delivered as C source files. This way, any Linux/Unix system that can compile C sources can be used as a host. The user has the option to directly include the C sources into his project or to build first an appropriate bdiAccess library.

The following source files are part of the bdiAccess distribution (in bdiaccux.zip):

bdiifc.c, bdiifc.h	The C source of the bdiAccess functions
bdilnk.c, bdilnk,h	The C source of the data link functions
compress.c, compress.h	The C source of the compression algorithm
bdicmd.h, bdierror.h	Additional C header files used by the bdiAccess modules
myapp.c	A simple demo that uses bdiAccess functions
makefile	A make file to build the siple demo application

Following an example of a make file to build a bdiAccess based application:

```
CFLAGS       = -g
LDFLAGS      =
OBJ          = myapp.o bdiifc.o bdilnk.o compress.o
all:         myapp
myapp:       $(OBJ)
             $(CC) $(LDFLAGS) $^ -o $@
myapp.o:     myapp.c bdierror.h bdicmd.h bdilnk.h bdiifc.h elf.h
bdiifc.o:    bdiifc.c bdierror.h bdicmd.h bdilnk.h bdiifc.h
bdilnk.o:    bdilnk.c bdierror.h bdicmd.h bdilnk.h compress.h
compress.o:  compress.c compress.h
```

BDI Setup Tool:

In order to update the firmware / logic of the BDI and to set the network parameters, a setup tool is included for Linux/Unix hosts. In the ZIP Archive bdisetup.zip are all sources to build this utility. More information about this utility can be found in the bdisetup.c source file.

To build the setup utility use: gcc bdisetup.c bdidll.c -o bdisetup

Configuration Example:

bdisetup -v -p/dev/ttyS0 -b57
bdisetup -u -p/dev/ttyS0 -b57 -aACC -tMPC800
bdisetup -c -p/dev/ttyS0 -b57 -i151.120.25.101

2. BDI Access Library Functions

Following a description of the bdiAccess functions.

2.1. Communication Functions

2.1.1. BDI_Connect()

Windows Hosts:
Opens the communication channel between the PC and the BDI. The channel to use is defined as a string.

For communication via a serial communication port use the following initialization string:
"COMx baudrate"

baudrate	enter the baudrate (9600, 19200, 38400, 57600, 115200) e.g. "COM1 57600"

For communication via an Ethernet use the following initialisation string:
"NETWORK ipaddress [port]"

ipaddress	enter the IP address of the BDI in the form xxx.xxx.xxx.xxx or a name if there exists an appropriate entry in the HOSTS file.
port	enter the bdiNet port where the BDI-HS is connected (1 or 2) For BDI this parameter is optional (must be 1 if supplied). e.g. "NETWORK 151.120.25.101 1"

```
/****************************************************************************
 ****************************************************************************

    Opens the connection to the BDI and connects to the firmware.

    Parameter Strings: "channel parameter1 parameter2 ... parameterN"
    e.g.:  COM1 at 38400baud   : "COM1 38400"
           COM1 at 115kBaud    : "COM2 115200"
           Via bdiNet Port 1   : "NETWORK 151.120.25.100 1"

    INPUT  : parameter      a string with the parameters
    OUTPUT : RETURN         0 if okay or a negativ number if error.

 ****************************************************************************/

int BDI_Connect(const char* parameter);
```

Examples:

```
result = BDI_Connect("COM1 57600");

result = BDI_Connect("NETWORK 151.120.25.100 1");
```

Linux/Unix Hosts:

Opens the communication channel between the Linux/Unix host and the BDI.

```
/*************************************************************************
 *************************************************************************

    BDI_Connect:

     Opens the connection to the BDI.

     INPUT  : host      a string for the comm port "/dev/..." or IP number
              param     the baudrate for serial connection
                        the port number for UDP/IP connection
     OUTPUT : RETURN    0 if okay or a negativ number if error.

 *************************************************************************/

int BDI_Connect(const char *host, DWORD param)
```

host	Defines the communication channel: /dev/..... for serial connection IP number for ethernet connection
param	An additional parameter. Baudrate for a serial connection (9600,19200,38400,57600 or 115200). UDP port for an ethernet connection. Use always 2001.

Examples:

```
result = BDI_Connect("/dev/ttyS0", 57600);

result = BDI_Connect("192.168.1.7", 2001);

result = BDI_Connect("bdi2000", 2001);    /* bdi2000 is in the hosts file */
```

2.1.2. BDI_Disconnect()

Closes the connection between the PC and the BDI.

```
/*************************************************************************
 *************************************************************************

    Closes the connection to the BDI.

    INPUT  : -
    OUTPUT : RETURN         0 if okay or a negativ number if error.

 *************************************************************************/

int BDI_Disconnect(void);
```

2.2. Register Access Functions

2.2.1. BDI_RegisterGet()

This function reads one or multiple target registers. Which register to read is defined in a list. Every list entry has two fields, the register type and the register address.

```
typedef struct {
  WORD      regType;
  WORD      regAddr;
} BDI_RegTypeT;

/***************************************************************************
 ***************************************************************************

    Read target registers

    INPUT:  count       the number of registers to read
            pRegList    points to the list with the registers to read
    OUTPUT: pValues     points to the array where to store the results
            return      0 if okay or a negativ number if error

 ***************************************************************************/

int BDI_RegisterGet(WORD count, const BDI_RegTypeT* pRegList, DWORD* pValues);
```

Example:
The following example reads two registers.

```
static BDI_RegTypeT     regList[] = {BDI_RT_PPC_GPR, 0, BDI_RT_PPC_GPR, 1};
static DWORD            regValue[2];
result = BDI_RegisterGet(2, regList, regValue);
```

2.2.2. BDI_RegisterSet()

This function writes one or multiple target registers. Which register to write is defined in a list. Every list entry has two fields, the register type and the register address. The value(s) to write are stored in an array.

```
/***************************************************************************
 ***************************************************************************

    Write to target registers

    INPUT:  count       the number of registers to write
            pRegList    points to the list with the registers to write
            pValues     points to the array with the values to write
    OUTPUT: return      0 if okay or a negativ number if error

 ***************************************************************************/

int BDI_RegisterSet(WORD count, const BDI_RegTypeT* pRegList, DWORD* pValues);
```

Example:
The following example writes to the SSR0 register.

```
static BDI_RegTypeT     regSSR0 = {BDI_RT_PPC_SPR, 26};
static DWORD            newPC;
result = BDI_RegisterSet(1, &regSSR0, & newPC);
```

All 16bit registers are always transferred as 32bit values.

Register Types and Addresses:

PowerPC Targets (e.g. MPC860):

Register Type	Register Address	Accessed Target Register
BDI_RT_PPC_GPR	0 ... 31	General-purpose registers
BDI_RT_PPC_SPR*	0 ... 1023	Special-purpose registers
BDI_RT_PPC_CR	-	Condition Register
BDI_RT_PPC_FPSCR	-	Floating-point Status and Control Register
BDI_RT_PPC_MSR	-	Machine State Register
BDI_RT_PPC_MM8**	0 ... 4095	MPC8xx: 8 bit memory mapped register
BDI_RT_PPC_MM16**	0 ... 4095	MPC8xx: 16 bit memory mapped register
BDI_RT_PPC_MM32**	0 ... 4095	MPC8xx: 32 bit memory mapped register
BDI_RT_PPC_IAR*	-	Instruction Address Register (PC)
BDI_RT_PPC_SR	0 ... 15	Segment registers
BDI_RT_PPC_DCR	0 ... 1023	PPC4xx: Device-control registers
BDI_RT_PPC_PMR	0 ... 1023	MPC85xx: Performance Monitor registers

* For targets with BDM interface (MPC8xx/MPC5xx) IAR accesses register SRR0 (SPR 26).
** Not recommended: Only supported for MPC8xx targets.

ColdFire Targets (e.g. MCF5206):

Register Type	Register Address	Accessed Target Register
BDI_RT_MCF_DATA	0 ... 7	Data registers
BDI_RT_MCF_ADDR	0 ... 7	Address registers
BDI_RT_MCF_CTRL	0x002	Cache Control Register (CACR)
BDI_RT_MCF_CTRL	0x004	Access Control Unit 0 (ACR0)
BDI_RT_MCF_CTRL	0x005	Access Control Unit 1 (ACR1)
BDI_RT_MCF_CTRL	0x801	Vector Base Register (VBR)
BDI_RT_MCF_CTRL	0x80E	Status Register (SR)
BDI_RT_MCF_CTRL	0x80F	Program Counter (PC)
BDI_RT_MCF_CTRL	0x000...0xFFF	other control registers, see ColdFire manual

CPU32 Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_CPU32_DATA	0 ... 7	Data Registers
BDI_RT_CPU32_ADDR	0 ... 7	Address Registers
BDI_RT_CPU32_PC	-	Program Counter
BDI_RT_CPU32_SR	-	Status Register (32bit transfer)
BDI_RT_CPU32_USP	-	User Stack Pointer
BDI_RT_CPU32_SSP	-	Supervisor Stack Pointer
BDI_RT_CPU32_VBR	-	Vector Base Register
BDI_RT_CPU32_SFC	-	Source Function Code Register
BDI_RT_CPU32_DFC	-	Destination Function Code Register

CPU16 Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_CPU16_D	-	Register D
BDI_RT_CPU16_E	-	Register E
BDI_RT_CPU16_IX	-	Index Register X
BDI_RT_CPU16_IY	-	Index Register Y
BDI_RT_CPU16_IZ	-	Index Register Z
BDI_RT_CPU16_K_EXT	-	Address Extension (K) Register
BDI_RT_CPU16_CCR	-	Condition Code Register [15:4]
BDI_RT_CPU16_SP	-	Stack Pointer (includes extension SK)
BDI_RT_CPU16_PC	-	Program Counter (includes extension PK)
BDI_RT_MAC_HR	-	MAC Register H
BDI_RT_MAC_IR	-	MAC Register I
BDI_RT_MAC_AM	-	MAC Register AM [31:0]
BDI_RT_MAC_SL_AM	-	MAC Register SL and AM[35:32]
BDI_RT_MAC_XM_YM	-	MAC Register XM:YM

CPU12 Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_CPU12_D	-	Register D
BDI_RT_CPU12_IX	-	Index Register X
BDI_RT_CPU12_IY	-	Index Register Y
BDI_RT_CPU12_CCR	-	Condition Code Register
BDI_RT_CPU12_SP	-	Stack Pointer
BDI_RT_CPU12_PC	-	Program Counter

ARM / XScale Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_ARM_GPR	0 ... 15	Current Register Set
BDI_RT_ARM_CPSR	-	Current Program Status Register
BDI_RT_ARM_USR	8 ... 14	User / System Registers
BDI_RT_ARM_FIQ	8 ... 14	FIQ Banked Registers
BDI_RT_ARM_SPSR_FIQ	-	FIQ Saved Program Status Register
BDI_RT_ARM_SVC	13,14	Supervisor Banked Registers
BDI_RT_ARM_SPSR_SVC	-	Supervisor Saved Program Status Register
BDI_RT_ARM_ABT	13,14	Abort Banked Registers
BDI_RT_ARM_SPSR_ABT	-	Abort Saved Program Status Register
BDI_RT_ARM_IRQ	13,14	IRQ Banked Registers
BDI_RT_ARM_SPSR_IRQ	-	IRQ Saved Program Status Register
BDI_RT_ARM_UND	13,14	Undefined Banked Registers
BDI_RT_ARM_SPSR_UND	-	Undefined Saved Program Status Register
BDI_RT_ARM_ICEB	0 ... 31	ICEBreaker registers. Normally the BDI handles this registers. The debugger should only access the Debug Comms Registers (address 4 and 5).
BDI_RT_ARM_CP15	See below	CP15 registers
BDI_RT_ARM_CPn	See below	CP registers, n = 0 ... 14 (only XScale and ARM11)

M-CORE Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_MCORE_GPR	0 ... 15	General-purpose registers
BDI_RT_MCORE_AFR	0 ... 15	Alternate file registers
BDI_RT_MCORE_PC	-	Program Counter
BDI_RT_MCORE_PSR	-	Processor Status Register
BDI_RT_MCORE_CR	1	VBR
BDI_RT_MCORE_CR	2	EPSR
BDI_RT_MCORE_CR	3	FPSR
BDI_RT_MCORE_CR	4	EPC
BDI_RT_MCORE_CR	5	FPC
BDI_RT_MCORE_CR	6	SS0
BDI_RT_MCORE_CR	7	SS1
BDI_RT_MCORE_CR	8	SS2
BDI_RT_MCORE_CR	9	SS3
BDI_RT_MCORE_CR	10	SS4
BDI_RT_MCORE_CR	11	GCR
BDI_RT_MCORE_CR	12	GSR
BDI_RT_MCORE_ONCE*	4	Memory Breakpoint Counter A (MBCA)
BDI_RT_MCORE_ONCE*	5	Memory Breakpoint Counter B (MBCB)
BDI_RT_MCORE_ONCE*	6	Program Counter Fifo (see M-CORE manual)
BDI_RT_MCORE_ONCE*	7	Breakpoint Address Base Register A (BABA)
BDI_RT_MCORE_ONCE*	8	Breakpoint Address Base Register B (BABB)
BDI_RT_MCORE_ONCE*	9	Breakpoint Address Mask Register A (BAMA)
BDI_RT_MCORE_ONCE*	10	Breakpoint Address Mask Register B (BAMB)
BDI_RT_MCORE_ONCE*	13	OnCE Control Register (OCR)
BDI_RT_MCORE_ONCE*	14	OnCE Status Register (OSR) (read only)

* The debugger can access some OnCE registers in order to fully have control over the built in memory breakpoint logic. As soon as the OCR is written with a non zero value, the BDI assumes, that the debugger on the host handles hardware breakpoints.

MIPS Targets:

Register Type	Register Address	Accessed Target Register
BDI_RT_MIPS_GPR	0 ... 31	General-purpose registers
BDI_RT_MIPS_PC	-	Program Counter
BDI_RT_MIPS_HI	-	HI register
BDI_RT_MIPS_LO	-	LO register
BDI_RT_MIPS_CP0	0 ... 31 | 0x0s00	CP0 registers s : defines the used "Select" (0 ... 7)
BDI_RT_MIPS_CP1	0 ... 31	CP1 control registers
BDI_RT_MIPS_FPR	0 ... 31	Floating-Point registers (only 32-bit)

CPn register number format for ARM / Xscale targets:

The numbers for CP registers have a special numbering scheme which depends on the ARM CPU type. More information is also found in the ARM documentation.

ARM710T, ARM720T,ARM740T:
The 16bit register number is used to build the appropriate MCR/MRC instruction to access the CP15 register.

```
+-----+-+-------+-------+-------+
|opc_2|0|  CRm  |0 0 0 0|  nbr  |
+-----+-+-------+-------+-------+
```

Normally opc_2 and CRm are zero and therefore you can simply enter the CP15 register number.

ARM920T:
Via JTAG, CP15 registers are accessed either direct (physical access mode) or via interpreted MCR/MRC instructions. Read also ARM920T manual, part "Debug Support - Scan Chain 15".

Register number for physical access mode (bit 12 = 0):

```
+-----+-+-----+-+-----+-+-------+
|0 0 0|0|0 0 0|i|0 0 0|x|  nbr  |
+-----+-+-----+-+-----+-+-------+
```

The bit "i" selects the instruction cache (scan chain bit 33), the bit "x" extends access to register 15 (scan chain bit 38).

Register number for interpreted access mode (bit 12 = 1):

```
+-----+-+-------+-----+-+-------+
|opc_2|1|  CRm  |opc_1|0|  nbr  |
+-----+-+-------+-----+-+-------+
```

The 16bit register number is used to build the appropriate MCR/MRC instruction.

ARM940T, ARM946E, ARM966E:
The CP15 registers are directly accessed via JTAG.

```
+-----+-+-----+-+-----+-+-------+
|0 0 0|0|0 0 0|i|0 0 0|x|  nbr  |
+-----+-+-----+-+-----+-+-------+
```

The bit "i" selects the instruction cache (scan chain bit 32), the bit "x" extends access to register 6 (scan chain bit 37).

ARM926E:
The 16bit register number contains the fields of the appropriate MCR/MRC instruction that would be used to access the CP15 register.

```
+-+-----+-+-----+-------+-------+
|-|opc_1|-|opc_2|  CRm  |  nbr  |
+-+-----+-+-----+-------+-------+
```

Normally opc_1, opc_2 and CRm are zero and therefore you can simply enter the CP15 register number.

TI925T:
The CP15 registers are directly accessed via JTAG. The following table shows the numbers used to access the CP15 registers and functions:

```
 0 (or 0x30) : ID
 1 (or 0x31) : Control
 2 (or 0x32) : Translation table base
 3 (or 0x33) : Domain access control
 5 (or 0x35) : Fault status
 6 (or 0x36) : Fault address
 8 (or 0x38) : Cache information
13 (or 0x3d) : Process ID

0x10 : TI925T Status
0x11 : TI925T Configuration
0x12 : TI925T I-max
0x13 : TI925T I-min
0x14 : TI925T Thread ID

0x18 : Flush I+D TLB
0x19 : Flush I TLB
0x1a : Flush I TLB entry
0x1b : Flush D TLB
0x1c : Flush D TLB entry

0x20 : Flush I cache
0x22 : Flush I cache entry
0x23 : Flush D cache
0x24 : Flush D cache entry address
0x25 : Clean D cache entry address
0x26 : Clean + Flush D cache entry address
0x27 : Flush D cache entry index
0x28 : Clean D cache entry index
0x29 : Clean + Flush D cache entry index
0x2a : Clean D cache
0x2b : Drain Write buffer

0x37 : I cache TLB Lock-Down
0x3a : D cache TLB Lock-Down
```

XScale, ARM11:
The register number is used to build the appropriate MCR or MRC instruction.

```
+-----+-+-------+-----+-+-------+
|opc_2|0|  CRm  |opc_1|0|  nbr  |
+-----+-+-------+-----+-+-------+
```

Some examples:

CP15 : ID register (CRn = 0, opcode_2 = 0) ➔ 0x0000

CP15 : Cache Type (CRn = 0, opcode_2 = 1) ➔ 0x2000

CP15 : Invalidate I cache line (CRn = 7, opcode_2 = 1, CRm = 5) ➔ 0x2507

2.2.3. BDI_FPRegGet()

This function reads one or multiple floating-point registers.

```
/****************************************************************************
 ****************************************************************************

    Read from floating-point register(s)

    INPUT:  first       the first register to read
            last        the last register to read
    OUTPUT: data        the read data (8 bytes for every register)
            return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FPRegGet(WORD first, WORD last, BYTE* data);
```

Note:
The register data is stored in big-endian format (MSB first).

Example:
The following example reads all floating-point registers of a PPC750.

```
static BYTE regValue[32 * 8];

result = BDI_FPRegGet(0, 31, regValue);
```

2.2.4. BDI_FPRegSet()

This function writes to one or multiple floating-point registers.

```
/****************************************************************************
 ****************************************************************************

    Write to floating-point register

    INPUT:  first       the first register to write
            last        the last register to write
            data        data to write (8 bytes for every register)
    OUTPUT: return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FPRegSet(WORD first, WORD last, BYTE* data);
```

Note:
The register data must be stored in big-endian format (MSB first).

Example:
The following example write to floating-point register 13.

```
static BYTE regValue[8];

regValue[0] = ....;
      .....
regValue[7] = ....;
result = BDI_FPRegGet(13, 13, regValue);
```

2.2.5. BDI_VRegGet()

This function reads one or multiple vector registers.

```
/****************************************************************************
 ****************************************************************************

    Read from vector register(s)

    INPUT:  first       the first register to read
            last        the last register to read
    OUTPUT: data        the read data (16 bytes for every register)
            return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_VRegGet(WORD first, WORD last, BYTE* data);
```

Note:
The register data is stored in big-endian format (MSB first).

Example:
The following example reads all vector registers of a MPC7400.

```
static BYTE regValue[32 * 16];

result = BDI_VRegGet(0, 31, regValue);
```

2.2.6. BDI_VRegSet()

This function writes to one or multiple vector registers.

```
/****************************************************************************
 ****************************************************************************

    Write to vector register(s)

    INPUT:  first       the first register to write
            last        the last register to write
            data        data to write (16 bytes for every register)
    OUTPUT: return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_VRegSet(WORD first, WORD last, BYTE* data);
```

Note:
The register data must be stored in big-endian format (MSB first).

Example:
The following example writes to vector register 13.

```
static BYTE regValue[16];

regValue[00] = ....;
      .....
regValue[15] = ....;
result = BDI_VRegGet(13, 13, regValue);
```

2.3. Memory Access Functions

2.3.1. BDI_GetByte(), BDI_GetWord(), BDI_GetLong()

Read a 8bit, 16bit or 32bit value from the target memory.

```
/****************************************************************************
 ****************************************************************************

    Reads a target byte,word,long

    INPUT:  addr         address to read from
    OUTPUT: value        the read value
            return       0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_GetByte(DWORD addr, BYTE  *value);
int BDI_GetWord(DWORD addr, WORD  *value);
int BDI_GetLong(DWORD addr, DWORD *value);
```

2.3.2. BDI_SetByte(), BDI_SetWord(), BDI_SetLong()

Write a 8bit, 16bit or 32bit value to the target memory.

```
/****************************************************************************
 ****************************************************************************

    Writes to a target byte,word,long

    INPUT:  addr         address to write to
            value        value to write
    OUTPUT: return       0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_SetByte(DWORD addr, BYTE  value);
int BDI_SetWord(DWORD addr, WORD  value);
int BDI_SetLong(DWORD addr, DWORD value);
```

2.3.3. BDI_DumpBlock()

Read a block of data from the target memory.

```
/***************************************************************************
 ***************************************************************************

    Reads a block of bytes from the target memory

    INPUT:  addr            address to read from
            count           number of bytes to read (up to 64k)
    OUTPUT: block           the read data block
            return          error code

 ***************************************************************************/

int BDI_DumpBlock(DWORD addr, WORD count, BYTE *block);
```

2.3.4. BDI_DumpFile()

Reads a block of target memory and stores it in a S-Record file.

```
/***************************************************************************
 ***************************************************************************

    Reads a block of bytes from the target memory and stores it in a
    S-Record file.

    INPUT:  fileName        the full path and filename of the S-record file
            addr            address to read from
            count           number of bytes to read
    OUTPUT: return          0 if okay or a negativ number if error

 ***************************************************************************/

int BDI_DumpFile(const char *fileName, DWORD addr, DWORD  count);
```

2.3.5. BDI_DumpBinary()

Reads a block of target memory and stores it in a binary file.

```
/***************************************************************************
 ***************************************************************************

    Reads a block of bytes from the target memory and stores it in a
    binary file.

    INPUT:  fileName        the full path and filename of the binary file
            addr            address to read from
            count           number of bytes to read
    OUTPUT: return          0 if okay or a negativ number if error

 ***************************************************************************/

int BDI_DumpBinary(const char *fileName, DWORD addr, DWORD  count);
```

2.3.6. BDI_LoadBlock()

Write a block of data to the target memory.

```
/****************************************************************************
 ****************************************************************************

    Writes a block to the target RAM memory

    INPUT:  addr            address to write to
            count           number of bytes in the block (up to 64k)
            block           the data block
    OUTPUT: errorAddr       address of the failing byte
            return          error code

 ****************************************************************************/

int BDI_LoadBlock(DWORD addr, WORD count, BYTE *block, DWORD *errorAddr)
```

2.3.7. BDI_LoadFile(), BDI_LoadFile2()

Writes a S-Record file into the target RAM memory.

```
/****************************************************************************
 ****************************************************************************

    Writes a S-Record file into the target RAM memory

    INPUT:  fileName        the full path and filename of the S-record file
            offset          the offset added to build the load address
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_LoadFile(const char *fileName, DWORD *errorAddr)
int BDI_LoadFile2(const char *fileName, DWORD offset, DWORD *errorAddr)
```

2.3.8. BDI_LoadBinary()

Writes a Binary file into the target RAM memory.

```
/****************************************************************************
 ****************************************************************************

    Writes a Binary file into the target RAM memory

    INPUT:  fileName        the full path and filename of the binary file
            startAddr       the target address of the binary image
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_LoadBinary(const char *fileName, DWORD startAddr, DWORD *errorAddr)
```

2.3.9. BDI_VerifyBlock()

Compares a block of data against the target memory.

```
/****************************************************************************
 ****************************************************************************

    Compares a data block against the target memory

    INPUT:  addr            address of the target memory
            count           number of bytes in the block (up to 64k)
            block           the data block
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_VerifyBlock(DWORD addr, WORD count, BYTE *block, DWORD *errorAddr);
```

2.3.10. BDI_VerifyFile(), BDI_VerifyFile2()

Compares the content of a S-Record file against the target memory.

```
/****************************************************************************
 ****************************************************************************

    Compares a S-Record file against the target memory

    INPUT:  fileName        the full path and filename of the S-record file
            offset          the offset added to build the load address
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_VerifyFile(const char *fileName, DWORD *errorAddr);
int BDI_VerifyFile2(const char *fileName, DWORD offset, DWORD *errorAddr);
```

2.3.11. BDI_VerifyBinary()

Writes a Binary file into the target RAM memory.

```
/****************************************************************************
 ****************************************************************************

Compares a Binary file against the target memory

    INPUT:  fileName        the full path and filename of the binary file
            startAddr       the target address of the binary image
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_VerifyBinary(const char *fileName, DWORD startAddr, DWORD *errorAddr)
```

2.4. Target Control Functions

2.4.1. BDI_TargetReset()

Resets the target. A real physical reset is executed (Reset Line is driven low). After a reset, normally some basic initializations must be done before the target memory can be accessed (e.g. init the memory controller, disable watchdog, ...). See also the function *BDI_TargetStartup()* which resets and initializes the target with one function call.

```
/****************************************************************************
 ****************************************************************************

    Reset the target.

    INPUT:   -
    OUTPUT: return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_TargetReset(void);
```

Note for HC12 targets:
Before using this function, BDM communication speed has to be setup either with *BDI_SetBdmSpeed()* or *BDI_TargetStartup()* .

2.4.2. BDI_SetBdmSpeed()

The BDI selects the BDM communication speed based on the current CPU clock. If this function defines a CPU clock rate that is higher than the real clock rate, BDM communication may fail. When selecting a clock rate slower than possible, BDM communication still works but not as fast as possible.

```
/****************************************************************************
 ****************************************************************************

    Sets the speed for the BDM communication.

    The new speed is used until the next target reset

    INPUT  : cpuClock    the clock rate of the target processor
    OUTPUT : return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_SetBdmSpeed(DWORD cpuClock);
```

Note for HC12 targets:
This function should be called before *BDI_ResetTarget()* to tell the BDI which BDM communication clock to use (see also chapter 2.7). The parameter cpuClock is used as follows:

cpuClock — This defines the BDM communication clock:

BDI-HS:	BDI1000:		
0 = ECLK signal	0 = ECLK	10 = 5.5 MHz	20 = 2.0 MHz
1 = 8 MHz	1 = 24 MHz	11 = 5.3 MHz	21 = 1.7 MHz
2 = 4 MHz	2 = 16 MHz	12 = 4.8 MHz	22 = 1.5 MHz
3 = 2 MHz	3 = 12 MHz	13 = 4.4 MHz	23 = 1.2 MHz
4 = 1 MHz	4 = 11 MHz	14 = 4.0 MHz	24 = 1.0 MHz
5 = 500 kHz	5 = 9.6 MHz	15 = 3.7 MHz	
6 = 250 kHz	6 = 8.0 MHz	16 = 3.0 MHz	
7 = 125 kHz	7 = 7.3 MHz	17 = 2.7 MHz	
	8 = 6.8 MHz	18 = 2.4 MHz	
	9 = 6.0 MHz	19 = 2.2 MHz	

Example for a CPU32 target:
The following example reset the target, makes some basic initializations and then adjust the BDM communication speed.

```
int result;
result = BDI_ResetTarget();
result = BDI_SetByte(0xFFFFFA21, 0x04);       /* SYPCR: watch dog */
result = BDI_SetWord(0xFFFFFA04, 0x7F00);     /* SYNCR: 16.7MHz   */
result = BDI_SetLong(0xFFFFFA48, 0x00077870); /* CSBOOT: 1wait, r/w, both */
result = BDI_SetLong(0xFFFFFA4C, 0x10055070); /* CS0: RAM write (MSB)*/
result = BDI_SetLong(0xFFFFFA58, 0x10053070); /* CS3: RAM write (LSB)*/
result = BDI_SetLong(0xFFFFFA60, 0x10056870); /* CS5: RAM read (word access) */
result = BDI_SetBdmSpeed(16777000);
```

2.4.3. BDI_SetOption()

This function allows to transfer a string option to the BDI. The meaning of the string option depends on the target connected to the BDI.

```
/****************************************************************************
 ****************************************************************************

    Writes an option string to the BDI

    INPUT  : id           the option ID
             data         the option string
    OUTPUT : return       0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_SetOption(BYTE id, const char* data);
```

Note for ARM targets:
This function is used to set the “Scan Init” (id = 1) and “Scan Post” (id = 2) strings.
This low level JTAG sequences maybe used to configure the TI ICEpick module.

Commands supported in the SCANINIT and SCANPOST strings:

```
I<n>=<...b2b1b0>  write IR, b0 is first scanned
D<n>=<...b2b1b0>  write DR, b0 is first scanned
                    n  : the number of bits 1..256
                    bx : a data byte, two hex digits
W<n>              wait for n (decimal) micro seconds
T1                assert TRST
T0                release TRST
R1                assert RESET
R0                release RESET
CH<n>             clock TCK n (decimal) times with TMS high
CL<n>             clock TCK n (decimal) times with TMS low
```

Example of an ICEPick configuration:

```
// Configure ICEPick module to make ARM926 TAP visible
static const char* scanInit =
"r0:w10000:"
"r1:t1:w1000:t0:w1000:" // assert reset and toggle TRST
"i6=07:d8=89:i6=02:"    // connect and select router
"d32=81000082:"         // set IP control
"d32=a018206f:"         // configure TAP0
"d32=a018216f:cl5:"     // enable TAP0, clock 5 times in RTI
"i10=ffff";             // scan bypass


// download Scan Init string
result = BDI_SetOption(BDI_OPT_ARM_SCANINIT, scanInit);
```

The scan init string is then used during the *BDI_TargetStartup()* call.

2.4.4. BDI_TargetStartup()

This function resets and initializes the target. After forcing a physical hardware reset, the initialization list is processed and then the BDM clock rate is adjusted (see JTAG note).

The init list is an array of the following structure:

```
typedef struct {
  WORD    command;
  DWORD   address;
  DWORD   value;
} BDI_InitTypeT;


/****************************************************************************
 ****************************************************************************

    Reset and startup target with an init list

    INPUT:  resetTime     reset time in milliseconds
            cpuClock      CPU clock in hertz after the init list is processed
            initCount     number of entries in the init list (max. 128)
            pInitList     points to the initialisation list
    OUTPUT: RETURN        0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_TargetStartup(      WORD           resetTime,
                            DWORD          cpuClock,
                            WORD           initCount,
                      const BDI_InitTypeT *pInitList);
```

Details about the parameters are specified on the following pages.
If initCount is zero, the entries in the local Init List are used (see next page).

Example for MPC8xx targets:
The following example resets the target for 100ms and makes some basic initializations for an MPC860 target. After processing the init list, the BDI assumes that the target runs at 20MHz.

```
int           result;
BDI_InitTypeT initList[] = {
  {BDI_INIT_PPC_WSPR, 638,        0x02200000}, // IMMR
  {BDI_INIT_PPC_WSPR, 158,        0x00000007}, // ICTRL
  {BDI_INIT_PPC_WM32, 0x02200000, 0x01632440}, // SIUMCR
  {BDI_INIT_PPC_WM32, 0x02200004, 0xFFFFFF88}, // SYPCR
  {BDI_INIT_PPC_WM16, 0x02200200, 0x0002    }, // TBSCR
  {BDI_INIT_PPC_WM32, 0x02200320, 0x55CCAA33}, // RTCSCK
  {BDI_INIT_PPC_WM16, 0x02200220, 0x0102    }, // RTCSC
  {BDI_INIT_PPC_WM16, 0x02200240, 0x0002    }  // PTSCR
};

result = BDI_TargetStartup(100,      // 100ms reset time
                           20000000, // 20MHz
                           sizeof initList / sizeof initList[0],
                           initList);
```

Alternate way to define the Init List:

For environments where it is difficult to handle the pInitList parameter (pointer to Array of Structures) there exists a different way to define the init list. Via multiple calls to *BDI_InitListAdd()* it is possible to define the init list before calling *BDI_TargetStartup()* with an initCount of zero and a NULL pointer for pInitList.

Clear the local Init List:

```
int BDI_InitListReset(void);
```

Add an entry to the local Init List:

```
int BDI_InitListAdd(WORD  command, DWORD address, DWORD value);
```

The example on the previous page maybe replaced with:

```
int           result;

result = InitListReset();

result = InitListAdd(BDI_INIT_PPC_WSPR, 638,        0x02200000);
result = InitListAdd(BDI_INIT_PPC_WSPR, 158,        0x00000007);
result = InitListAdd(BDI_INIT_PPC_WM32, 0x02200000, 0x01632440);
result = InitListAdd(BDI_INIT_PPC_WM32, 0x02200004, 0xFFFFFF88);
result = InitListAdd(BDI_INIT_PPC_WM16, 0x02200200, 0x0002    );
result = InitListAdd(BDI_INIT_PPC_WM32, 0x02200320, 0x55CCAA33);
result = InitListAdd(BDI_INIT_PPC_WM16, 0x02200220, 0x0102    );
result = InitListAdd(BDI_INIT_PPC_WM16, 0x02200240, 0x0002    );

result = BDI_TargetStartup(100,      // 100ms reset time
                           20000000, // 20MHz
                           0,        // use prepeared init list
                           NULL);
```

Note for HC12 targets:
For HC12 targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

ramAddr	regAddr	cpu type	clock

clock — This defines the BDM communication clock:

BDI-HS:	BDI1000:		
0 = ECLK signal	0 = ECLK	10 = 5.5 MHz	20 = 2.0 MHz
1 = 8 MHz	1 = 24 MHz	11 = 5.3 MHz	21 = 1.7 MHz
2 = 4 MHz	2 = 16 MHz	12 = 4.8 MHz	22 = 1.5 MHz
3 = 2 MHz	3 = 12 MHz	13 = 4.4 MHz	23 = 1.2 MHz
4 = 1 MHz	4 = 11 MHz	14 = 4.0 MHz	24 = 1.0 MHz
5 = 500 kHz	5 = 9.6 MHz	15 = 3.7 MHz	
6 = 250 kHz	6 = 8.0 MHz	16 = 3.0 MHz	
7 = 125 kHz	7 = 7.3 MHz	17 = 2.7 MHz	
	8 = 6.8 MHz	18 = 2.4 MHz	
	9 = 6.0 MHz	19 = 2.2 MHz	

cpuType — Defines the target CPU type:

0 = HC812A4	1 = HC912B32	2 = HC912D60
3 = HC912DG128	4 = HC912DA128	5 = HC912D60A
6 = HC912DG128A	7 = HC912DT128A	
8 = HCS12 256k	9 = HCS12 128k	10 = HCS12 64k
11 = HCS12 32k	12 = <reserved>	13 = HCS12 512k
14 = HCS12 E256	15 = HCS12 E128	
16 = S12X 128K	17 = S12X 256K	18 = S12X 512K
19 = S12X 768K	20 = S12X 1024K	

regAddr — Defines the upper byte of the register block address

ramAddr — Defines the upper byte of the internal SRAM address

Note for ARM7/9/9E targets:
For ARM targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	endian	cpu type	clock

clock — Defines the JTAG clock rate:

	BDI3000	BDI2000	BDI1000
0 =	Adaptive	Adaptive	Adaptive
1 =	32 MHz	16 MHz	6 MHz
2 =	16 MHz	8 MHz	3 MHz
3 =	11 MHz	4 MHz	1 MHz
4 =	8 MHz	1 MHz	500 kHz
5 =	5 MHz	500 kHz	200 kHz
6 =	4 MHz	200 kHz	100 kHz
7 =	1 MHz	100 kHz	50 kHz
8 =	500 kHz	50 kHz	20 kHz
9 =	200 kHz	20 kHz	10 kHz
10 =	100 kHz	10 kHz	
11 =	50 kHz	5 kHz	
12 =	20 kHz	2 kHz	
13 =	10 kHz	1 kHz	
14 =	5 kHz		
15 =	2 kHz		
16 =	1 kHz		

cpu type — Defines the target CPU type:

0 = ARM7TDMI	8 = TMS470
1 = ARM7DI	9 = ARM9E
2 = ARM710T	10 = ARM946E
3 = ARM720T	11 = ARM966E
4 = ARM740T	12 = TI925T
5 = ARM9TDMI	13 = MAC7100
6 = ARM920T	14 = ARM926E
7 = ARM940T	

endian — Defines the target endian :
0 = little endian
1 = big endian

Example: big endian, ARM720T, 8 MHz ==> cpuClock = 0x00010302

Note for ARM11 targets:
For ARM targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	endian	cpu type	clock

clock — Defines the JTAG clock rate:

	BDI3000	BDI2000
0 =	Adaptive	Adaptive
1 =	32 MHz	16 MHz
2 =	16 MHz	8 MHz
3 =	11 MHz	4 MHz
4 =	8 MHz	1 MHz
5 =	5 MHz	500 kHz
6 =	4 MHz	200 kHz
7 =	1 MHz	100 kHz
8 =	500 kHz	50 kHz
9 =	200 kHz	20 kHz
10 =	100 kHz	10 kHz
11 =	50 kHz	
12 =	20 kHz	
13 =	10 kHz	
14 =	5 kHz	
15 =	2 kHz	
16 =	1 kHz	

cpu type — Defines the target CPU type:
0 = ARM1136

endian — Defines the target endian :
0 = little endian
1 = big endian

Example: big endian, ARM1136, 8 MHz ==> cpuClock = 0x00010002

Note for XScale targets:
For ARM targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	endian	cpu type	clock

clock — Defines the JTAG clock rate:

	BDI3000	BDI2000	BDI1000
0 =	32 MHz	16 MHz	5.5 MHz
1 =	16 MHz	8 MHz	2.8 MHz
2 =	11 MHz	4 MHz	1.8 MHz
3 =	8 MHz	1 MHz	1.4 MHz
4 =	5 MHz	500 kHz	
5 =	4 MHz	200 kHz	
6 =	1 MHz	100 kHz	
7 =	500 kHz		
8 =	200 kHz		
9 =	100 kHz		
10 =	50 kHz		
11 =	20 kHz		
12 =	10 kHz		

cpu type — Defines the target CPU type:

0 = IOP310, 80200
1 = IOP321
2 = PXA200
3 = IXP400, IOP331, PXA270
4 = IXP2300, IOP340 (Manzano)
5 = PXA280 (Monahans)

endian — Defines the target endian :

0 = little endian
1 = big endian

Example: big endian, PXA250, 8 MHz ==> cpuClock = 0x00010201

Note for PPC4xx targets:
For PPC4xx targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	-	cpu type	clock

clock — Defines the JTAG clock rate:

	BDI3000	BDI2000	BDI1000
0 =	32 MHz	16 MHz	5.5 MHz
1 =	16 MHz	8 MHz	2.8 MHz
2 =	11 MHz	4 MHz	1.8 MHz
3 =	8 MHz	1 MHz	1.4 MHz
4 =	5 MHz	500 kHz	
5 =	4 MHz	200 kHz	
6 =	1 MHz	100 kHz	
7 =	500 kHz		
8 =	200 kHz		
9 =	100 kHz		
10 =	50 kHz		
11 =	20 kHz		
12 =	10 kHz		
13 =	5 kHz		

cpu type — Defines the target CPU type:

0 = 401
1 = 403
2 = 405
3 = 440 (for 440GX define IR length 8 via an init list entry)

Example: PPC405, 8 MHz ==> cpuClock = 0x00000201

Note for PPC6xx/7xx/82xx/83xx targets:
For this targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	-	cpu type	clock

cpuClock Defines the JTAG clock rate:

	BDI3000	BDI2000
0 =	32 MHz	16 MHz
1 =	16 MHz	8 MHz
2 =	11 MHz	4 MHz
3 =	8 MHz	
4 =	5 MHz	
5 =	4 MHz	

cpu type Defines the target CPU type:

0 = PPC603e
1 = PPC603ev
2 = PPC750/740
3 = MPC8260
4 = MPC8240
5 = PPC750CX
6 = MPC7400
7 = MPC4200
8 = PPC750FX/GX
9 = MPC8280/8270
10 = MGT5200
11 = MPC8220
12 = MPC83xx (e300)

Example: MPC7400, 8 MHz ==> cpuClock = 0x00000601

Note for MPC744x/745x/8641 targets:
For this targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	-	cpu type	clock

cpuClock Defines the JTAG clock rate:

	BDI3000	BDI2000
0 =	32 MHz	16 MHz
1 =	16 MHz	8 MHz
2 =	11 MHz	4 MHz
3 =	8 MHz	
4 =	5 MHz	
5 =	4 MHz	

cpu type Defines the target CPU type:

0 = MPC7450
1 = MPC7455/45
2 = MPC7457/47
3 = MPC7447A
4 = MPC7448
5 = MPC8641
6 =
7 =
8 =
9 =

Example: MPC7448, 8 MHz ==> cpuClock = 0x00000401

Note for MPC85xx targets:
For this targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	-	cpu type	clock

cpuClock Defines the JTAG clock rate:

	BDI3000	BDI2000
0 =	32 MHz	16 MHz
1 =	16 MHz	8 MHz
2 =	11 MHz	4 MHz
3 =	8 MHz	1 MHz
4 =	5 MHz	500 kHz
5 =	4 MHz	200 kHz
6 =	1 MHz	100 kHz
7 =	500 kHz	50 kHz
8 =	200 kHz	20 kHz
9 =	100 kHz	10 kHz
10 =	50 kHz	5 kHz
11 =	20 kHz	
12 =	10 kHz	
13 =	5 kHz	

cpu type Defines the target CPU type:
0 = MPC8560
1 = MPC8540
2 = MPC8548

Example: MPC8560, 8 MHz ==> cpuClock = 0x00000001

Note for M-CORE targets:
For M-Core targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	-	cpu type	clock

clock Defines the JTAG clock rate:

BDI2000:	BDI1000:
0 = 16.6 MHz	0 = 5.5 MHz
1 = 8.3 MHz	1 = 2.8 MHz
2 = 5.5 MHz	2 = 1.8 MHz
3 = 4.1 MHz	3 = 1.4 MHz

cpu type Defines the target CPU type:
0 = MMC2001
1 = MMC2107
2 = MMC2114

Example: MMC2107, 8 MHz ==> cpuClock = 0x00000101

Note for MIPS32 targets:
For MIPS32 targets, the parameter cpuClock is used for not only for clock information but also to define some additional information the BDI needs.

-	endian	cpu type	clock

clock — Defines the JTAG clock rate:

	BDI3000	BDI2000
0 =	32 MHz	16 MHz
1 =	16 MHz	8 MHz
2 =	11 MHz	5 MHz
3 =	8 MHz	4 MHz
4 =	5 MHz	
5 =	4 MHz	

cpu type — Defines the target CPU type:

0 = RC32300
1 = Au1000
2 = M4K
3 = M4KE
4 = M24K

endian — Defines the target endian :

0 = little endian
1 = big endian

Example: big endian, M4Kc, 8 MHz ==> cpuClock = 0x00010201

Initlist Commands:

PowerPC Targets (e.g. MPC860):

Command	Address	Value	Description
BDI_INIT_PPC_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. A delay may be used after programming the clock synthesiser to let the VCO lock again before proceeding. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_PPC_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_PPC_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_PPC_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_PPC_WGPR	0..31	value	Write to a general-purpose register
BDI_INIT_PPC_WSPR	0..1023	value	Write to a special-purpose register
BDI_INIT_PPC_WDCR	0..1023	value	PPC400: Write to a device-control register
BDI_INIT_PPC_WMSR	-	value	Write to the Machine State Register
BDI_INIT_PPC_WCR	-	value	Write to the Condition Register
BDI_INIT_PPC_SUPM	MCR address	MDR address	This entry is used set the addresses of the Memory Command Register (MCR) and Memory Data Register (MDR). These registers are used to program the UPM (for more information see MPC860 user's manual). The field Address should get the address of the MCR and the field Value should get the address of the MDR. (e.g. Address = 0x02200168, Value = 0x0220017c)
BDI_INIT_PPC_WUPM	MCR value	MDR value	This writes an entry into the UPM RAM array. The value in the Value field is first written to the MDR, then the value in the Address field is written to the MCR.
BDI_INIT_PPC_TSZ1 BDI_INIT_PPC_TSZ2 BDI_INIT_PPC_TSZ4 BDI_INIT_PPC_TSZ8	Start	end	COP targets (PPC6xx/7xx/82xx/83xx): Defines an address range with a limited transfer size. Normally the BDI tries to access the memory with a transfer size of 8 bytes.
BDI_INIT_PPC_MMAP	Start	end	PPC400 and COP targets (PPC6xx/7xx/82xx/83xx): In order to avoid invalid memory accesses via the JTAG interface, it is possible to define up to 32 memory ranges where access is allowed. If no range is defined at all, the whole 4GB address range is enabled for access
BDI_INIT_PPC_WTLB	Start	end	PPC440: Add a TLB entry. On a PPC440, the MMU is always active. Therefore it is necessary to add the appropriate TLB entries before memory can be accessed.

Note for MPC82xx targets:
The UPM of a MPC82xx is setup different than the one in the MPC860.
The SUPM and WUPM init list entries have a different meaning.
SUPM <UPM memory addr> <MDR addr>
WUPM <don’t care> <UPM data>
UPM write example (UPM memory @ 0x10000000) :

```
WM32  0x0471011C  0xFF000000  ;OR3
WM32  0x04710118  0x10000081  ;BR3
WM32  0x04710170  0x10000000  ;MAMR : setup for array write
SUPM  0x10000000  0x04710188  ;set address of UPM range and MDR
WUPM  0x00000000  0xaba00000  ;write UPM array
WUPM  0x00000000  0xaba00001
      ...
WUPM  0x00000000  0xaba0003E
WUPM  0x00000000  0xaba0003F
WM32  0x04710170  0x00000000  ;MAMR : setup for normal mode
```

Note for PPC6xx/7xx/82xx/83xx and MPC744x/745x/8641 targets:
In order to change some special configuration parameters of the BDI, the SPR entry is used. Normal PPC SPR's covers a range from 0 to 1023. Other SPR's are used to set BDI internal registers:

8001 For slow memory it may be necessary to increase the number of clocks used to execute a memory access cycle. If for example you cannot access boot ROM content with the default configuration of your memory controller, define additional memory access clocks with this SPR number in the init list.

8002 Defines an alternate boot address. Normally a PPC boots from 0xFFF00100. A MPC8260 has also the option to boot from 0x00000100. The BDI needs to know the boot address in order to set the correct hardware breakpoint during startup.

8003 Defines the base address of the L2 cache private memory. Because L2 cache private memory cannot be accessed directly via JTAG, the BDI loads some support code into the workspace and uses it to access this memory range. Therefore a workspace is necessary to access this memory range.

8004 Defines the size of the L2 cache private memory in bytes (e.g. 0x100000 for 1Mbyte).

8005 Defines the time in milliseconds after reset the BDI lets the target execute code. If this entry is present, the target is not forced to debug mode immediately out of reset. After forcing a HRESET the BDI let the target run for this time. This is useful if there is a monitor present that should setup the target.

8007 PPC6xx/7xx/82xx:
A value of 1 indicates that the PPC core operates in 32-bit data bus mode.
MPC744x/745x:
Write to this special register a value of 1 if no burst reads should be used.

8008 If the BDI should generate data parity bits when writing to memory, write to this special register a value of 1.

8009 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the COP-HRESET line and starting communicating with the target. This init list entry may be necessary if COP-HRESET is delayed on its way to the PowerPC reset pin.

8010 If the BDI should forces the QACK pin (pin 2) on the COP connector low, write to this special register a value of 1. By default the QACK pin is not driven by the BDI. This option maybe useful for PPC750 and PPC7400 targets.

The MPC83xx processors allows to override the Reset Configuration Word (RCW) via JTAG. The following two special configuration parameters define the RCW. Always define both words.

8020 Reset Configuration Word High (RCWH).
8021 Reset Configuration Word Low (RCWL).

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the PowerPC chip. Enter the appropriate information also for the devices following the PowerPC chip.

8011 Number of JTAG devices connected before the PowerPC chip.
8012 Total IR length of the JTAG devices connected before the PowerPC chip.
8013 Number of JTAG devices connected after the PowerPC chip.
8014 Total IR length of the JTAG devices connected after the PowerPC chip.

Note for PPC4xx targets:
In order to change some special configuration parameters of the BDI, the SPR entry is used. Normal PPC SPR's covers a range from 0 to 1023. Other SPR's are used to set BDI internal registers:

8005 By default the BDI forces a system reset via the JTAG debug interface. With this entry in the init list the reset type can be changed. The following values are supported (see also PPC4xx manuals): 0 = NONE, 1 = CORE, 2 = CHIP, 3 = SYSTEM (default).
8005 A PPC4xx target can be forced to debug mode in two different ways. With this entry in the init list this mode can be selected a follows: 0 = JTAG stop command (default), 1 = Assert HALT pin.
8006 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This delay is necessary when a target needs some wake-up time after a reset.
8007 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This delay is necessary when a target needs some wake-up time after a reset.
8010 By default the BDI assumes that the HALT signal is low active at the debug connector. In cases where this signal is high active, enter a value of 1 for this special configuration parameter.

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the PPC4xx chip. Enter the appropriate information also for the devices following the PPC4xx chip.

8001 Number of JTAG devices connected before the PPC4xx chip.
8002 Total IR length of the JTAG devices connected before the PPC4xx chip.
8003 Number of JTAG devices connected after the PPC4xx chip.
8004 Total IR length of the JTAG devices connected after the PPC4xx chip.

8008 The IR value for the device(s) connected after the device under test. Only the last 8 bits can be defined. Default is 0xFF (bypass). Useful for Xilinx Virtex-II Pro chips.
8009 The length of the PPC4xx IR register (default is 7). For 440GX/EP/SP use 8.

Adding entries to the PPC440 TLB:
For PPC440 targets, it is necessary to setup the TLB before memory can be accessed. This is because on a PPC440 the MMU is always enabled. The init list entry WTLB allows an initial setup of the TLB array. The first WTLB entry clears also the whole TLB array.

The epn parameter defines the effective page number, space, size and WIMG flags:

```
+----------------------+-+-+----+----+
|          EPN         |-|S|SIZE|WIMG|
+----------------------+-+-+----+----+
            22          1 1   4    4
```

The rpn parameter defines the real page number and access rights:

```
+----+------------------------+------+
|ERPN|          RPN           |XWRXWR|
+----+------------------------+------+
   4             22               6
```

Not all fields of a TLB entry are defined with the above values. The other values except the valid bit are implicit set to zero. The XWRXWR field starts with the user access rights. See also PPC440 user’s manual part "Memory Management".

The following example clears the TLB and adds two entries to access ROM and SDRAM:

```
WTLB 0xF0000095 0x1F00003F ;Boot Space 256MB, cache inhibited, guarded
WTLB 0x00000098 0x0000003F ;SDRAM 256MB @0x00000000, write-through
```

MPC85xx Targets:

Command	Address	Value	Description
BDI_INIT_PQ3_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. A delay may be used after programming the clock synthesiser to let the VCO lock again before proceeding. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_PQ3_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_PQ3_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_PQ3_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_PQ3_WM64	address	value	Write a double word (64bit) to the selected memory place. This entry is mainly used to unlock flash blocks. The pattern written is generated by duplicating the value: (0x12345678 -> 0x1234567812345678).
BDI_INIT_PQ3_WGPR	0..31	value	Write to a general-purpose register
BDI_INIT_PQ3_WSPR	0..1023	value	Write to a special-purpose register
BDI_INIT_PQ3_WMSR	-	value	Write to the Machine State Register
BDI_INIT_PQ3_WCR	-	value	Write to the Condition Register
BDI_INIT_PQ3_SUPM	MEM addr	MDR addr	Starts a sequence of writes to the UPM RAM array: MEM addr: an address in the UPM memory range MDR addr: the address of the MDR register
BDI_INIT_PQ3_WUPM	Don't care	data	Write to the UPM RAM array asfollows: *mdraddr = data, *memaddr = 0
BDI_INIT_PQ3_TSZ1 BDI_INIT_PQ3_TSZ2 BDI_INIT_PQ3_TSZ4 BDI_INIT_PQ3_TSZ8	Start	End	Defines an address range with a limited transfer size. Normally when the BDI reads or writes a memory block, it tries to access the memory with a burst access. The TSZx entry allows to define a maximal transfer size for up to 8 address ranges.
BDI_INIT_PQ3_MMAP	Start	End	In order to avoid invalid memory accesses via the JTAG interface, it is possible to define up to 32 memory ranges where access is allowed. If no range is defined at all, the whole 4GB address range is enabled for access
BDI_INIT_PQ3_EXEC	addr	Time (ms)	This entry cause the processor to start executing the code at addr. The second parameter defines a time in ms how long the BDI let the processor run until it is halted. By default the BDI let it run for 500 us. This EXEC function maybe used to create TLB entries via some helper code.

Note:
In order to change some special configuration parameters of the BDI, the SPR entry is used. Normal PPC SPR's covers a range from 0 to 1023. Other SPR's are used to set BDI internal registers:

8002 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the COP-HRESET line and starting communicating with the target. This init list entry may be necessary if COP-HRESET is delayed on its way to the PowerPC reset pin.

8003 Defines the time in milliseconds after reset the BDI lets the target execute code. If this entry is present, the target is not forced to debug mode immediately out of reset. After forcing a HRESET the BDI let the target run for this time. This is useful if there is a monitor present that should setup the target. In the special case when this entry is present but with a value of zero, the BDI does not use the L2SRAM for a boot loop but does simple stop the core out of reset with a core halt command.

8004 This parameter defines how the BDI accesses memory via JTAG. By default it uses the System Access Port (SAP). Alternatively the BDI can use the e500 core to access memory. Enter a value of 0 if the BDI should use the e500 core to access memory. A value of 1 (default) uses the System Access Port (SAP) to access memory.

HC12 Targets:

Command	Address	Value	Description
BDI_INIT_CPU12_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. A delay may be used after programming the clock synthesiser to let the VCO lock again before proceeding. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_CPU12_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_CPU12_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_CPU12_WM32	address	value	Write a 32bit value to target memory

M-CORE Targets:

Command	Address	Value	Description
BDI_INIT_MMC_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_MMC_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_MMC_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_MMC_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_MMC_WGPR	0..15	value	Write to a general-purpose register
BDI_INIT_MMC_WAFR	0..15	value	Write to a alternate file register
BDI_INIT_MMC_WCR	0 ..12	value	Write to a control register
BDI_INIT_MMC_WPC	-	value	Write to the program counter

MIPS32 Targets:

Command	Address	Value	Description
BDI_INIT_MIPS_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_MIPS_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_MIPS_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_MIPS_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_MIPS_WGPR	0..31	value	Write to a general-purpose register
BDI_INIT_MIPS_WCP0	number	value	Write to a CP0 Register. Number = 0 ... 31 | 0x0s00 s : defines the used "Select" (0 ... 7)

Special BDI Configuration Registers:
In order to change some special configuration parameters of the BDI, the GPR entry is used. Normal MIPS GPR's covers a range from 0 to 31. Other GPR's are used to set BDI internal configuration registers.

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the MIPS chip. Enter the appropriate information also for the devices following the MIPS chip.

8001 Number of JTAG devices connected before the MIPS chip.
8002 Total IR length of the JTAG devices connected before the MIPS chip.
8003 Number of JTAG devices connected after the MIPS chip.
8004 Total IR length of the JTAG devices connected after the MIPS chip.

8005 By default, the BDI asserts the hardware reset pin during reset processing. After writing 0 or 1 to this special register, the BDI no longer drives RESET low. This may be useful in some special cases. If the value written is 1, then a reset via the EJTAG control register is forced. A value of 0 does not assert any reset at all.

8007 In order to speed-up dump and verify, a workspace in target RAM can be defined. The same address as for the programming algorithm maybe used.

8008 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This delay is necessary when a target needs some wake-up time after a reset.

ARM7/9/9E Targets:

Command	Address	Value	Description
BDI_INIT_ARM_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_ARM_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_ARM_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_ARM_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_ARM_WGPR	0..15	value	Write to a general-purpose register
BDI_INIT_ARM_WCPSR	-	value	Write to the Program Status Register
BDI_INIT_ARM_WCP15	regNbr	value	Write to a CP15 register
BDI_INIT_ARM_MMAP	start	end	In order to avoid invalid memory accesses via the JTAG interface, it is possible to define up to 32 memory ranges where access is allowed. If no range is defined at all, the whole 4GB address range is enabled for access
BDI_INIT_ARM_EXEC	addr	Time (ms)	This entry causes the processor to start executing the code at Address. The second parameter defines a time in ms how long the BDI lets the processor run until it is halted. By default (value = 0) the BDI lets it run for 500 us. This EXEC function maybe used to access CP15 registers that are not directly accessible via JTAG..

Special BDI Configuration Registers:
In order to change some special configuration parameters of the BDI, the GPR entry is used. Normal ARM GPR's covers a range from 0 to 15. Other GPR's are used to set BDI internal configuration registers.

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the ARM chip. Enter the appropriate information also for the devices following the ARM chip.

8001 Number of JTAG devices connected before the ARM chip.
8002 Total IR length of the JTAG devices connected before the ARM chip.
8003 Number of JTAG devices connected after the ARM chip.
8004 Total IR length of the JTAG devices connected after the ARM chip.

8005 This entry in the init list allows to change the JTAG clock frequency. This is useful if you have to start with a slow JTAG clock out of reset but after some initialization (e.g. PLL setup) you can use a faster clock. The value you enter selects the JTAG frequency as defined for the “cpuClock” parameter (see beginning of this chapter).

8006 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This delay is necessary when a target needs some wake-up time after a reset (e.g. Cirrus EP7209).

8007 By default, the BDI asserts the RESET signal during reset processing. After writing zero to this special register, the BDI no longer drives RESET low. This may be useful in some special cases.

8008 During JTAG debugging, the PC increments while the BDI stuffs instruction into the ARM core. It may be necessary to set the PC to a safe non-vector address before external memory is accessed to prevent pre-fetching code from an invalid address range. Enter a safe non-vector address for the PC into this special BDI registers.

8009 By default, the TRST signal is driven with an open-drain driver by the BDI. Write a 1 to this special BDI register if the TRST signal should be driven with a push-pull driver.

ARM11 Targets:

Command	Address	Value	Description
BDI_INIT_ARM11_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_ ARM11_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_ ARM11_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_ ARM11_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_ ARM11_WGPR	0..15	value	Write to a general-purpose register
BDI_INIT_ ARM11_WCPSR	-	value	Write to the Program Status Register
BDI_INIT_ ARM11_WCP15	regNbr	value	Write to a CP15 register
BDI_INIT_ ARM11_WCPn	regNbr	value	Write to a CP register, n = 0 ... 15

Special BDI Configuration Registers:
In order to change some special configuration parameters of the BDI, the GPR entry is used. Normal ARM GPR's covers a range from 0 to 15. Other GPR's are used to set BDI internal configuration registers.

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the ARM chip. Enter the appropriate information also for the devices following the ARM chip.

8001 Number of JTAG devices connected before the ARM chip.
8002 Total IR length of the JTAG devices connected before the ARM chip.
8003 Number of JTAG devices connected after the ARM chip.
8004 Total IR length of the JTAG devices connected after the ARM chip.

8005 This entry in the init list allows to change the JTAG clock frequency. This is useful if you have to start with a slow JTAG clock out of reset but after some initialization (e.g. PLL setup) you can use a faster clock. The value you enter selects the JTAG frequency as defined for the “cpuClock” parameter (see beginning of this chapter).

 ©ABATRON AG

8006 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This delay is necessary when a target needs some wake-up time after a reset.

8007 By default, the BDI asserts the RESET signal during reset processing. After writing zero to this special register, the BDI no longer drives RESET low. This may be useful in some special cases.

8008 This entry in the init list allows to define a time (in ms) the BDI asserts the hardware reset signal. By default the reset signal is asserted for about 3 ms.

8009 By default, the TRST signal is driven with an open-drain driver by the BDI. Write a 1 to this special BDI register if the TRST signal should be driven with a push-pull driver.

8010 When using adaptive clocking, write a 1 to this special BDI register if the ARM core is clocked with a frequency slower that 6 MHz.

XScale Targets:

Command	Address	Value	Description
BDI_INIT_XSC_DELAY	-	milliseconds	The defined amount of time is waited at this position when working through the init list. When selecting this option, enter the amount of time in milliseconds.
BDI_INIT_XSC_WM8	address	value	Write a 8bit value to target memory
BDI_INIT_XSC_WM16	address	value	Write a 16bit value to target memory
BDI_INIT_XSC_WM32	address	value	Write a 32bit value to target memory
BDI_INIT_XSC_WGPR	0..15	value	Write to a general-purpose register
BDI_INIT_XSC_WCPSR	-	value	Write to the Program Status Register
BDI_INIT_XSC_WCP15	regNbr	value	Write to a CP15 register
BDI_INIT_XSC_WCPn	regNbr	value	Write to a CP register, n = 0 ... 15

Special BDI Configuration Registers:
In order to change some special configuration parameters of the BDI, the GPR entry is used. Normal XScale GPR's covers a range from 0 to 15. Other GPR's are used to set BDI internal configuration registers.

The BDI can also handle systems with multiple devices connected to the JTAG scan chain. In order to put the other devices into BYPASS mode and to count for the additional bypass registers, the BDI needs some information about the scan chain layout. Enter the number and total instruction register (IR) length of the devices present before the ARM chip. Enter the appropriate information also for the devices following the ARM chip.

8001 Number of JTAG devices connected before the ARM chip.
8002 Total IR length of the JTAG devices connected before the ARM chip.
8003 Number of JTAG devices connected after the ARM chip.
8004 Total IR length of the JTAG devices connected after the ARM chip.

8005 Address of the Debug Handler, 2k aligned in the range 0x00000000 ... 0x01FEF800 or 0xFE000800 ... 0xFFFFF800.

8006 When this entry is present, the BDI fills the default vector table in the Mini IC (except the debug/reset vector) with the requested opcode. Also the default vector table in the Mini IC will not be updated each time before the target is restarted.

8007 When this entry is present, the BDI fills the relocated vector table in the Mini IC (except the debug/reset vector) with the requested opcode. Also the relocated vector table in the Mini IC will not be updated each time before the target is restarted.

8008 This entry in the init list allows to define a delay time (in ms) the BDI inserts between releasing the reset line and starting communicating with the target. This init list entry may be necessary if JTAG-RESET signal is delayed (e.g. routed through a reset chip) on its way to the XScale reset pin.

8009 By default, the TRST signal is driven with a push-pull driver by the BDI. Write a 0 to this special BDI register if the TRST signal should be driven with an open-drain driver.

8010 This entry in the init list allows to define a time (in ms) the BDI asserts the hardware reset signal. By default the reset signal is asserted for at least 500 ms.

2.4.5. BDI_TargetStart()

Start program execution.

```
/****************************************************************************
 ****************************************************************************

    Start the target CPU

    INPUT:   -
    OUTPUT: return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_TargetStart(void);
```

Example:
The following example sets the program counter of a MPC8xx/5xx and starts the target.

```
int TARGET_StartFrom(DWORD pc)
{
  BDI_RegTypeT    reg;

  reg.regType = BDI_RT_PPC_SPR;
  reg.regAddr = 26; /* SRR0 */
  error = BDI_RegisterSet(1, &reg, &pc);
  if (error == 0) error = BDI_TargetStart();
  return error;
} /* TARGET_WriteSPR */
```

2.4.6. BDI_TargetHalt()

Stop execution of target code.

```
/****************************************************************************
 ****************************************************************************

    Stop execution of target code

    INPUT:   -
    OUTPUT: return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_TargetHalt(void);
```

2.4.7. BDI_StateGet()

This function returns information about the current target state.

```
/****************************************************************************
 ****************************************************************************

    Get the current target state

    INPUT:   count       max. number of additional infos
    OUTPUT:  state       the current target state
             error       error that causes error state
             timeStamp   the reset time stamp
             count       number of additional information fields
             info        the additional infos
             return      0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_StateGet(  WORD*        state,
                   int*         error,
                   DWORD*       timeStamp,
                   WORD*        count,
                   DWORD*       info);
```

state	The current target state: 0 = reseted, no setup stored in BDI flash 1 = ready, startup successful passed, target in debug mode 2 = target is currently running 3 = error, target is in error state, needs reset command
error	If the current target state is 3 (error) this is the code of the last detected error.
timeStamp	Every time a target reset is processed, the current system time is stored. The system time is the number of seconds since the BDI has been started. This time stamp can be used to detect if the target has reseted during a debug session.
count	Returns the number of additional information fields. Before calling the function, load this parameter with the length of the info buffer.
info	Points to an array of DWORD's. The additional information is stored there. Information depends on target architecture.

Additional Information:

PowerPC MPC5xx/MPC8xx Targets:

Info1 The cause for the last change to debug mode

00 = <reserved>
01 = reset interrupt
02 = check stop interrupt
03 = machine check interrupt
04 = data storage interrupt
05 = instruction storage interrupt
06 = external interrupt
07 = alignment interrupt
08 = program interrupt
09 = floating-point unavailable interrupt
10 = decrementer interrupt
11 = <reserved>
12 = <reserved>
13 = system call interrupt
14 = trace interrupt (single step)
15 = floating-point assist interrupt
16 = <reserved>
17 = software emulation interrupt
18 = instruction TLB miss
19 = instruction TLB error
20 = data TLB miss
21 = data TLB error
22 - 27 = <reserved>
28 = load/store breakpoint (watchpoint)
29 = instruction breakpoint (watchpoint)
30 = external breakpoint (BDI forced debug mode entry)
31 = debug port NMI (nonmaskable debug entry request)
32 = software breakpoint raised

Info2 current program counter (PC)

Info3 The breakpoint address register (BAR)

Info4 The data address register (DAR)

Info5 The DSISR value

ColdFire Targets:

Info1	The cause for the last change to debug mode 0 = startup 1 = exception trapped by BDI 2 = BGND instruction in user code 3 = breakpoint raised 4 = watchpoint raised 5 = single step 6 = halt forced by BDI 7 = double bus fault
Info2	The current program counter (PC)
Info3	The current SR
Info4	The current frame pointer (A6)
Info5	The exception vector number

CPU32 Targets:

Info1	The cause for the last change to debug mode 0 = startup 1 = exception trapped by BDI 2 = BGND instruction in user code 3 = breakpoint raised 4 = watchpoint raised 5 = single step 6 = halt forced by BDI 7 = double bus fault
Info2	The current program counter (PC)
Info3	The current SR
Info4	The current frame pointer (A6)
Info5	The exception vector number

CPU16 Targets:

Info1	The cause for the last change to debug mode 0 = startup 1 = exception trapped by BDI 2 = BGND instruction in user code 3 = breakpoint raised 4 = watchpoint raised 5 = single step 6 = halt forced by BDI 7 = double bus fault
Info2	The current program counter (IP)
Info3	The current CCR
Info4	The current stack pointer (IZ)
Info5	The exception vector number

CPU12 Targets:

Info1	The cause for the last change to debug mode 0 = startup 1 = <reserved> 2 = BGND instruction in user code 3 = breakpoint raised 4 = watchpoint raised 5 = single step 6 = halt forced by BDI 7 = <reserved>
Info2	The current program counter (PC)

M-CORE Targets:

Info1	The cause for the last change to debug mode 00 = <reserved> 01 = <reserved> 02 = <reserved> 03 = <reserved> 04 = <reserved> 05 = trace occured (single step) 06 = software breakpoint raised 07 = hardware breakpoint raised 08 = halt forced by BDI 09 = hardware debug request 10 = external breakpoint request (BKREQ)
Info2	The current program counter (PC)
Info3	The current program status register (PSR)
Info4	The current stack pointer (R0)
Info5	The exception vector number

ARM Targets:

Info1	The cause for the last change to debug mode 00 = breakpoint raised 01 = watchpoint raised 02 = halt forced by BDI 03 = single step
Info2	The current program counter (R15)
Info3	The current program status register (CPSR)
Info4	The current link register (R14)
Info5	The current stack pointer (R13)
Info6	The last exception vector (currently not supported)
Info7	The target endianess. 0 = little , >0 = big

PowerPC PPC6xx/PPC7xx/MPC82xx/83xx and MPC744x/745x/8641 Targets:

Info1 The cause for the last change to debug mode

00 = <reserved>
01 = reset interrupt
02 = machine check interrupt
03 = data storage interrupt
04 = instruction storage interrupt
05 = external interrupt
06 = alignment interrupt
07 = program interrupt
08 = floating-point unavailable interrupt
09 = decrementer interrupt
10 = <reserved>
11 = <reserved>
12 = system call interrupt
13 = trace interrupt (single step)
14 = floating-point assist interrupt
15 = <reserved>
16 = instruction translation miss
17 = data load translation miss
18 = data load translation miss
19 = instruction address breakpoint
20 = system management interrupt
21 - 31 = <reserved>
32 = software breakpoint raised
33 = COP halt (BDI forced debug mode entry)
34 = COP freeze (BDI forced debug mode entry)

Info2 The current program counter (IAR)

Info3 The condition register (CR)

Info4 The machine state register (MSR)

Info5 The link register (LR)

PPC4xx Targets:

Info1	The cause for the last change to debug mode 0 = single step 1 = branch take 2 = exception 3 = trap 4 = unconditional debug event 5 = instruction breakpoint 6 = data breakpoint 7 = JTAG stop (BDI forced debug mode entry) 8 = software breakpoint
Info2	The current IAR (PC)
Info3	The current CR
Info4	The current MSR
Info5	The current LR
Info6	Additional information depends on halt entry cause (Info1) 2 = Exception vector 5,6 = Watchpoint ID that causes debug event

MIPS Targets:

Info1	The cause for the last change to debug mode 0 = single step 1 = software breakpoint 2 = data load breakpoint 3 = data store breakpoint 4 = instruction breakpoint 5 = halt forced by BDI 6 = exception trapped by BDI
Info2	The current program counter (PC)
Info3	The current status register (SR)
Info4	The current link register (R31)
Info5	The current stack pointer (R29)
Info6	Additional information depends on halt entry cause (Info1) 2,3,4 = Watchpoint ID that causes debug entry 6 = Exception Program Counter (EPC)
Info7	The target endianess. 0 = little , >0 = big

XScale Targets:

Info1	The cause for the last change to debug mode 0 = reset 1 = instruction breakpoint 2 = data breakpoint 3 = BKPT instruction (software breakpoint) 4 = halt forced by BDI 5 = exception trapped by BDI 6 = trace buffer full 7 = <reserved> 8 = single step
Info2	The current program counter (PC)
Info3	The current program status register (CPSR)
Info4	The current link register (R14)
Info5	The current stack pointer (R13)
Info6	The exception vector if info1 = 4
Info7	The target endianess. 0 = little , >0 = big

MPC85xx Targets:

Info1	The cause for the last change to debug mode 0 = unknown 1 = single step 2 = branch take 3 = exception 4 = trap 5 = unconditional debug event 6 = instruction breakpoint 7 = data breakpoint 8 = COP halt request 9 = COP stop request 10 = software breakpoint
Info2	The current IAR (PC)
Info3	The current CR
Info4	The current MSR
Info5	The current LR
Info6	The current CCSR

ARM11 Targets:

Info1	The cause for the last change to debug mode 0 = halt forced by BDI 1 = instruction breakpoint 2 = data breakpoint 3 = BKPT instruction (software breakpoint) 4 = external debug request 5 = exception trapped by BDI 6 = data abort 7 = instruction abort 8 = single step
Info2	The current program counter (PC)
Info3	The current program status register (CPSR)
Info4	The current link register (R14)
Info5	The current stack pointer (R13)
Info6	The exception vector if info1 = 4
Info7	The target endianess. 0 = little , >0 = big

2.5. Flash Programming Functions

2.5.1. BDI_ FlashSetType()

Defines the flash type used on the target system. Before any other flash related function can be used, this function must be called. The information about the flash type is stored within the access library and is used when programming or erasing flash memory.

NOTE: For HC12 targets, use the function *BDI_FlashSetupHC12().*

NOTE: For MPC555 internal flash, use the function *BDI_FlashSetup555().*

```
/***************************************************************************
 ***************************************************************************

    Defines the  Flash memory

    INPUT:  flashType       the used Flash Memory Type
            flashSize       the size of one flash memory chip in bytes
            pageSize        the size in bytes of a page of one chip
                            enter 0 if the chip has no pages
            memoryWidth     the memory bus width in bits (8,16,32)
            workSpace       some programming algorithm must run in the target
                            CPU, enter the base address of a RAM area reseved
                            for this algorithm.
    OUTPUT: return          0 if okay or a negativ number if error

 ***************************************************************************/

int BDI_FlashSetType(WORD  flashType,
                     DWORD flashSize,
                     WORD  pageSize,
                     WORD  memoryWidth,
                     DWORD workSpace);
```

flashType select the flash programming algorithm (see also bdiifc.h)

2 = Am29F010 family
5 = i28F001BX / i28F016SA family in x8 mode
6 = i28F400BX / i28F016SA family in x16 mode
11 = Am29F100 family in x8 mode
12 = Am29F100 family in x16 mode
22 = Atmel AT49 byte mode only chip
23 = Atmel AT49 byte/word chip in x8 mode
24 = Atmel AT49 byte/word chip in x16 mode
29 = Intel Strata flash in x8 mode
30 = Intel Strata flash in x16 mode
31 = AMD MirrorBit Flash byte mode only chip
32 = AMD MirrorBit Flash byte/word chip in byte mode
33 = AMD MirrorBit Flash byte/word chip in word mode
34 = Intel algorithm for x32 chip
35 = AMD x32 chip in x16 mode
36 = AMD x32 chip in x32 mode
37 = AMD Am29BDD160G chip in x16 mode (only for PPC5xx/8xx)
38 = AMD Am29BDD160G chip in x32 mode (only for PPC5xx/8xx)
39 = ColdFire Flash Module (CFM)
40 = MAC7100 Program Flash Module (CFM32)
41 = MAC7100 Data Flash Module (CFM16)

42 = Philips LPC2000 Flash Module
43 = ST STA2051 / STR71x internal flash
44 = Analog Device ADuC7000 internal flash
45 = ST ST30F7xx internal flash
46 = Atmel AT91SAM7Sxx
47 = New MirrorBit Flash byte/word chip in byte mode
48 = New MirrorBit Flash byte/word chip in word mode

flashSize	the size of one flash chip in bytes e.g. Am29F010 = 0x20000 = 131072 (128kByte)
pageSize	Currently not used, enter 0.
memoryWidth	the width of the memory bus in bits (8,16,32,64)
workSpace	the base address of a RAM area that can be use when the programming algorithm runs on the target CPU. If you don't want to use target memory, enter 0xFFFFFFFF as the base address. This disables using the fast programming mode. Programming is still possible but it is slower. If used, the size of the RAM area should be at least 2kBytes and should be aligned to a 4 byte boundary.

Supported Flash Memories:
There are currently 3 standard flash algorithm supported. The AMD, Intel and Atmel AT49 algorithm. Almost all currently available flash memories can be programmed with one of this algorithm. The flash type selects the appropriate algorithm and gives additional information about the used flash.

8bit only flash:	AM29F0X0 (MIRROR), I28FBXB, AT49
8/16 bit flash in 8bit mode:	AM29FX00B (MIRRORX8), I28FBXB (STRATAX8), AT49X8
8/16 bit flash in 16bit mode:	AM29FX00W (MIRRORX16), I28FBXW (STRATAX16), AT49X16
16bit only flash:	AM29FX00W, I28FBXW,AT49X16
16/32 bit flash in 16bit mode:	AM29DX16
16/32 bit flash in 32 bit mode:	AM29DX32

The AMD and AT49 algorithm are almost the same. The only difference is, that the AT49 algorithm does not check for the AMD status bit 5 (Exceeded Timing Limits).
Only the AMD and AT49 algorithm support chip erase. Block erase is only supported with the AT49 algorithm. If the algorithm does not support the selected mode, sector erase is performed. If the chip does not support the selected mode, erasing will fail. The erase command sequence is different only in the 6th write cycle. Depending on the selected mode, the following data is written in this cycle (see also flash data sheets): 0x10 for chip erase, 0x30 for sector erase, 0x50 for block erase.
To speed up programming of Intel Strata Flash and AMD MirrorBit Flash, an additional algorithm is implemented that makes use of the write buffer. This algorithm needs a workspace, otherwise the standard Intel/AMD algorithm is used.

Following some examples:

Flash	x 8	x 16	x 32	Chipsize
Am29F010	AM29F0X0	-	-	0x20000
Am29F800T/B	AM29FX00B	AM29FX00W	-	0x100000
Am29DL323C	AM29FX00B	AM29FX00W		0x400000
Am29PDL128G	-	AM29DX16	AM29DX32	0x1000000
Am29LV320M	MIRRORX8	MIRRORX16	-	0x400000
Intel 28F032B3	I28FBXB	-	-	0x400000
Intel 28F400B	I28FBXB	I28FBXW	-	0x080000
Intel 28F320C3	-	I28FBXW	-	0x400000
Intel 28F640J3A	STRATAX8	STATAX16	-	0x800000
AT49BV040	AT49	-	-	0x80000
AT49BV1614	AT49X8	AT49X16	-	0x200000
SST39VF160	-	AT49X16	-	0x200000
M58BW016BT	-	-	I28FBXD	0x200000

Note:
Not all targets support all algorithm types.

Examples:

```
int     result;

/* i28F400BX word mode, 16bit bus */
result = BDI_FlashSetType(BDI_FLASH_I28FBXW,
                          0x80000,     /* flash size is 512kByte */
                          0,           /* no pages               */
                          16,          /* 16bit data bus         */
                          0xFFFFFFFF); /* do not use target RAM  */


/* Am29F016 32bit bus */
result = BDI_FlashSetType(BDI_FLASH_AM29F0X0,
                          0x200000,    /* flash size is 2MB      */
                          0,           /* no pages               */
                          32,          /* 32 data bus            */
                          0x02000000); /* use target algorithm   */
```

2.5.2. BDI_ FlashEraseSector()

This function erases one flash sector.
NOTE: For HC12 targets, use the function *BDI_FlashEraseHC12().*
NOTE: For MPC555 internal flash, use the function *BDI_FlashErase555().*

```
/****************************************************************************
 ****************************************************************************

    Erase a flash sector

    INPUT:  sectorAddr      Startaddress of the sector to erase
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashEraseSector(DWORD sectorAddr);
```

Note:
For the LPC2000 internal flash, the sectorAddr parameter has a different meaning. It is a bitmap of the sectors to erase. See also description of *BDI_FlashSetupLPC2000()* .
For the ST STA2051 (STR710) and ST30F7xx the sectorAddr value is directly written to the FCR1 register to select the sectors that should be erased.

2.5.3. BDI_ FlashErase()

This function erases a flash sector, a flash block or a flash chip. Erasing a block or the whole chip is only supported if the used flash memory supports it. Only the AMD and AT49 algorithm support chip erase. Block erase is only supported with the AT49 algorithm. If the algorithm does not support the selected mode, sector erase is performed. If the chip does not support the selected mode, erasing will fail.

```
/****************************************************************************
 ****************************************************************************

    Erase a flash sector, block or chip

    INPUT:  mode      Erase mode
            addr      Startaddress of the sector, block or chip
    OUTPUT: return    0 if okay or a negativ number if error

 ****************************************************************************/

#define BDI_ERASE_SECTOR        0
#define BDI_ERASE_BLOCK         1
#define BDI_ERASE_CHIP          2

int BDI_FlashErase(WORD mode, DWORD addr);
```

Note:
The erase command sequence is different only in the 6th write cycle. Depending on the selected mode, the following data is written in this cycle (see also flash data sheets):

0x10 for chip erase
0x30 for sector erase
0x50 for block erase

The AMD and AT49 algorithm are almost the same. The only difference is, that the AT49 algorithm does not check for the AMD status bit 5 (Exceeded Timing Limits).

2.5.4. BDI_ FlashWriteBlock()

Programs a data block into the target flash memory.

```
/****************************************************************************
 ****************************************************************************

    Programs a data block into the target flash memory

    INPUT:  addr            address of the target memory
            count           number of bytes in the block (up to 64k)
            block           the data block
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashWriteBlock(DWORD addr, WORD count, BYTE *block, DWORD *errorAddr);
```

2.5.5. BDI_ FlashWriteFile(), BDI_ FlashWriteFile2()

Programs a S-Record file into the target flash memory.

```
/****************************************************************************
 ****************************************************************************

    Programs a S-Record file into the target flash memory

    INPUT:  fileName        the full path and filename of the S-record file
            offset          the offset added to build the load address
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashWriteFile(const char *fileName, DWORD *errorAddr);
int BDI_FlashWriteFile2(const char *fileName, DWORD offset, DWORD *errorAddr);
```

2.5.6. BDI_ FlashWriteBinary()

Programs a Binary file into the target flash memory.

```
/****************************************************************************
 ****************************************************************************

    Programs a Binary file into the target flash memory

    INPUT:  fileName        the full path and filename of the S-record file
            startAddr       the target address of the binary image
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashWriteBinary(const char  *fileName,
                               DWORD  startAddr,
                               DWORD *errorAddr);
```

Note: This function is not supported for HC12 targets.

2.5.7. BDI_FlashSetupHC12()

Defines the flash type used on a HC12 target system. Before any other flash related function can be used, this function must be called. The information about the flash type is stored within the access library and is used when programming or erasing flash memory.
This function also brings the BDI firmware into a special programming mode. The normal mode after the BDI is powered up is used for debugging and needs some information about the target to be stored in the BDI flash memory (e.g. the target CPU clock frequency).

In programming mode, the BDI adds or changes the following features:

- **BDI-HS only**: BDM communication clock is automatically selected. If ECLK is present, external clocking mode is used. Otherwise the BDI selects one of its internal clocks.
- Target information stored in the BDI flash is ignored.
- Address translation in order to access external memory is not the same as in debug mode. See chapter "Additional information"

Therefore the function *BDI_FlashSetupHC12()* should be called immediately after connecting to the BDI. If not flash programming is the reason for using the bdiAccess library, see chapter "Additional information" about how to use bdiAccess functions when the BDI is in debugging mode.

```
/****************************************************************************
 ****************************************************************************

    Defines the  Flash memory for HC12 targets

    INPUT:  cpuType         the HC12 CPU type (see header file)
            flashType       the used Flash Memory Type (paged flag)
            memoryWidth     the memory bus width in bits (8,16,32)
            flashSize       the size of one flash memory chip in bytes
            workSpace       some programming algorithm must run in the target
                            CPU, enter the base address of a RAM area reseved
                            for this algorithm.
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

/* Flash Types */
#define BDI_FLASH_AM29F0X0    2   /* e.g. Am29F010, Am29F040              */
#define BDI_FLASH_I28FBXB     5   /* e.g. i28F001BX, i28F400BX (byte-mode) */
#define BDI_FLASH_I28FBXW     6   /* e.g. i28F400BX (word-mode)           */
#define BDI_FLASH_AM29FX00B   11  /* e.g. Am29F100 (byte-mode)            */
#define BDI_FLASH_AM29FX00W   12  /* e.g. Am29F100 (word-mode)            */
#define BDI_HC12_EEPROM       14  /* HC12 internal EEPROM                 */
#define BDI_HC12_FLASH        15  /* HC12 internal Flash                  */
#define BDI_HC12A_EEPROM      18  /* internal EEPROM of HC12A targets     */
#define BDI_HC12A_FLASH       19  /* internal Flash of HC12A targets      */
#define BDI_STAR12_EEPROM     20  /* internal EEPROM of STAR12 targets    */
#define BDI_STAR12_FLASH      21  /* internal Flash of STAR12 targets     */

/* CPU Types */
#define BDI_CPU_HC812A4       0
#define BDI_CPU_HC912B32      1
#define BDI_CPU_HC912D60      2
#define BDI_CPU_HC912DG128    3
#define BDI_CPU_HC912DA128    4
#define BDI_CPU_HC912D60A     5
#define BDI_CPU_HC912DG128A   6
#define BDI_CPU_HC912DT128A   7
#define BDI_CPU_MC9S12D256    8
#define BDI_CPU_MC9S12D128    9
   ... see bdiifc.h ....
```

```
int BDI_FlashSetupHC12(WORD  cpuType,
                       WORD  flashType,
                       WORD  memoryWidth,
                       DWORD flashSize,
                       DWORD workSpace);
```

cpuType	Select the HC12 CPU type 0 = HC812A4 · 5 = HC912D60A · 10 = MC9S12Dx64 1 = HC912B32 · 6 = HC912DG128A · 11 = MC9S12Dx32 2 = HC912D60 · 7 = HC912DT128A · xx = see bdiifc.h 3 = HC912DG128 · 8 = MC9S12Dx256 4 = HC912DA128 · 9 = MC9S12Dx128
flashType	Select the flash programming algorithm (see bdiifc.h) 2 = Am29F010 family 5 = i28F001BX / i28F016SA family in byte-mode 6 = i28F400BX / i28F016SA family in word-mode 11 = Am29F100 family in byte-mode 12 = Am29F100 family in word-mode 14 = HC12 internal EEPROM 15 = HC12 internal Flash 18 = HC12A internal EEPROM (e.g. HC912DT128) 19 = HC12A internal Flash (e.g. HC912DT128) 20 = STAR12 internal EEPROM (e.g. MC9S12DP256) 21 = STAR12 internal Flash (e.g. MC9S12DP256)
memoryWidth	The width of the memory bus in bits (8,16,32)
flashSize	The size of one flash chip in bytes e.g. Am29F010 = 0x20000 = 131072 (128kByte)
workSpace	The base address of the internal SRAM. The BDI needs some RAM to store the programming algorithm and the data to program.

2.5.8. BDI_FlashEraseHC12()

This function erases one external flash sector, one internal flash block or the internal EEPROM.

```
/****************************************************************************
 ****************************************************************************

    Erase HC12 internal or external Flash memory

    INPUT:  flashAddr       the base address of the flash memory
            flashSize       the size of the flash memory
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashEraseHC12(DWORD flashAddr, DWORD flashSize)
```

flashAddr	The base address of the flash sector to erase
flashSize	The size of flash to erase. This parameter is only used for HC12 internal flash or EEPROM memory.

2.5.9. BDI_EnableAutoProgHC12()

This function enables the automatic programming mode.

```
/****************************************************************************
 ****************************************************************************

    Enable HC12 automatic programming mode

    INPUT:  commPort        string that defines the communication port
            cpuType         HC12 CPU type (and paged flag)
            bdmClock        BDM clock
            regAddr         base address of the HC12 register block
            ramAddr         base address of the internal SRAM
            initCount       number of entries in the init list
            pInitList       init list
            flashErase      set if flash should be erased
            flashVerify     set if flash should be verified
            flashName       name of the S-record file with the flash data
            eepErase        set if EEPROM should be erased
            eepVerify       set if EEPROM should be verified
            eepName         name of the S-record file with the EEPROM data
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_EnableAutoProgHC12(const char           *commPort,
                                 WORD            cpuType,
                                 WORD            bdmClock,
                                 WORD            regAddr,
                                 WORD            ramAddr,
                                 WORD            initCount,
                           const BDI_InitTypeT *pInitList,
                                 BOOL            flashErase,
                                 BOOL            flashVerify,
                           const char           *flashName,
                                 BOOL            eepErase,
                                 BOOL            eepVerify,
                           const char           *eepName)
```

commPort — The communication port.
For serial use : "COMx baudrate"
For network use : "NETWORK ipaddress"

cpuType — Select the HC12 CPU type (add BDI_HC12_PAGED for paged flash file)

0 = HC812A4	5 = HC912D60A	10 = MC9S12Dx64
1 = HC912B32	6 = HC912DG128A	11 = MC9S12Dx32
2 = HC912D60	7 = HC912DT128A	
3 = HC912DG128	8 = MC9S12Dx256	
4 = HC912DA128	9 = MC9S12Dx128	

bdmClock — This defines the BDM communication clock:

BDI-HS:	BDI1000:		
0 = ECLK signal	0 = ECLK	10 = 5.5 MHz	20 = 2.0 MHz
1 = 8 MHz	1 = 24 MHz	11 = 5.3 MHz	21 = 1.7 MHz
2 = 4 MHz	2 = 16 MHz	12 = 4.8 MHz	22 = 1.5 MHz
3 = 2 MHz	3 = 12 MHz	13 = 4.4 MHz	23 = 1.2 MHz
4 = 1 MHz	4 = 11 MHz	14 = 4.0 MHz	24 = 1.0 MHz
5 = 500 kHz	5 = 9.6 MHz	15 = 3.7 MHz	
6 = 250 kHz	6 = 8.0 MHz	16 = 3.0 MHz	
7 = 125 kHz	7 = 7.3 MHz	17 = 2.7 MHz	
	8 = 6.8 MHz	18 = 2.4 MHz	
	9 = 6.0 MHz	19 = 2.2 MHz	

regAddr	The base address of the register block
ramAddr	The base address of the SRAM used as workspace
initCount	The number of entries in the init list
pInitList	Points to the table with the init list entries
flashErase	Defines if the flash should be erased before programming
flashVerify	Defines if the flash should be verified after programming
flashName	The name of the S-record file with the flash data. If the addresses in the file are paged, add the flag BDI_HC12_PAGED to the “cpuType” parameter. For linear addresses, the BDI support addresses that are bottom or top aligned in a 1MB address space. For a MC9S12DP256 the linear addresses may be from 0x00000 to 0x3FFFF or from 0xC0000 to 0xFFFFF.
eepErase	Defines if the EEPROM should be erased before programming
eepVerify	Defines if the EEPROM should be verified after programming
eepName	The name of the S-record file with the EEPROM data. A value of NULL disables EEPROM programming

Example:

```
BDI_InitTypeT initList[] = {
  {BDI_INIT_CPU12_WM8, 0x10, 0x00 }, /* INITRM: Map RAM to 0x0000 */
  {BDI_INIT_CPU12_WM8, 0xFF, 0x00 }, /* PPAGE: Select Flash Array 0 */
  {BDI_INIT_CPU12_WM8, 0xF5, 0x00 }, /* FEEMCR: Enable program of boot block */
  {BDI_INIT_CPU12_WM8, 0xFF, 0x02 }, /* PPAGE: Select Flash Array 1 */
  {BDI_INIT_CPU12_WM8, 0xF5, 0x00 }, /* FEEMCR: Enable program of boot block */
  {BDI_INIT_CPU12_WM8, 0xFF, 0x04 }, /* PPAGE: Select Flash Array 2 */
  {BDI_INIT_CPU12_WM8, 0xF5, 0x00 }, /* FEEMCR: Enable program of boot block */
  {BDI_INIT_CPU12_WM8, 0xFF, 0x06 }, /* PPAGE: Select Flash Array 3 */
  {BDI_INIT_CPU12_WM8, 0xF5, 0x00 }, /* FEEMCR: Enable program of boot block */
  {BDI_INIT_CPU12_WM8, 0xF1, 0x00 }, /* EEPROT: Unprotect all EEPROM blocks */
};

int           result;

result = BDI_EnableAutoProgHC12("COM12 57600",
                                BDI_CPU_HC912DA128 | BDI_HC12_PAGED,
                                6,         /* select 8MHz BDM clock */
                                0x0000,    /* register address */
                                0x0800,    /* SRAM address */
                                sizeof initList / sizeof initList[0],
                                initList,
                                TRUE,      /* erase flash */
                                TRUE,      /* verify flash */
                                "myFlash.sss",
                                TRUE,      /* erase EEPROM */
                                TRUE,      /* verify EEPROM */
                                "myEEPROM.sss");
```

2.5.10. BDI_DisableAutoProgHC12()

This function disables the HC12 automatic programming mode.

```
/****************************************************************************
 ****************************************************************************

    Disable HC12 automatic programming mode

    INPUT:  commPort        a string that defines the communication port
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_DisableAutoProgHC12(const char *commPort)
```

commPort	The communication port.	
	For serial use :	"COMx baudrate"
	For network use :	"NETWORK ipaddress"

2.5.11. BDI_FlashSetup555()

Defines the parameters for MPC555 internal flash programming. Before any other flash related function can be used, this function must be called. The information about the flash type is stored within the access library and is used when programming or erasing flash memory.

```
/****************************************************************************
 ****************************************************************************

    Defines the programming parameters for MPC555 flash programming

    INPUT:  flashType       the used Flash Memory Type (16 or 17)
            cpuClock        the target CPU clock in hertz
            workSpace       address of workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

/* Flash Type */
#define BDI_MPC555_FLASH      16  /* MPC555 flash array                  */
#define BDI_MPC555_SHADOW     17  /* MPC555 flash shadow information     */

int BDI_FlashSetup555(WORD flashType, DWORD cpuClock, DWORD workSpace);
```

flashType — Select the flash type:
16 = MPC555 internal flash
17 = MPC555 internal shadow flash.

cpuClock — The CPU clock frequency in Hertz. The BDI needs this parameter to calculate the program/erase pulse timing. It is important to setup the correct frequency otherwise the erase/programming pulses are not correct.

workSpace — This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

Example:

```
#define CLOCK           20000000
#define WORKSPACE       0x003FC000  // use internal SRAM-B

static BDI_InitTypeT initList[] = {
{BDI_INIT_PPC_WSPR, 638,        0x00000800}, // IMMR: @0x00000000, Flash enabled
{BDI_INIT_PPC_WSPR, 158,        0x00000007}, // ICTRL: not serialized, no show
{BDI_INIT_PPC_WM32, 0x002FC004, 0xFFFFFF01}  // SYPCR : disable watchdog
};

result = BDI_Connect("NETWORK BDI2000");
result = BDI_FlashSetup555(BDI_MPC555_FLASH, CLOCK, WORKSPACE);
result = BDI_TargetStartup(0,
                           TARGET_CLOCK,
                           sizeof initList / sizeof initList[0],
                           initList);
result = BDI_FlashErase555(0x00000000, 0xFFFC0000, FALSE, CLOCK, WORKSPACE);
result = BDI_FlashWriteFile("flash32.sss", &errorAddr);
result = BDI_VerifyFile("flash32.sss", &errorAddr);
....
```

2.5.12. BDI_FlashErase555()

This function erases one or multiple flash block(s) in the MPC555 internal flash.

```
/****************************************************************************
 ****************************************************************************

    Erase MPC555 internal Flash memory

    The parameter blockMask has the following layout.

    +---------------+---------------+---------------+---------------+
    |   Module A    |   Module B    |               |               |
    +---------------+---------------+---------------+---------------+
    |0|1|2|3|4|5|6|7|0|1|2|3|4|5|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
    +---------------+---------------+---------------+---------------+
    |31           24|23           16|15            8|7             0|

    If the appropriate bit is set, the block will be erased.

    INPUT:  flashBase       the base address of the first flash module
            blockMask       the mask with the flash block to erase
            clearCensor     if set the censor bits will be erased
            cpuClock        the target CPU clock in hertz
            workSpace       address of workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashErase555(DWORD flashBase,
                      DWORD blockMask,
                      BOOL  clearCensor,
                      DWORD cpuClock,
                      DWORD workSpace);
```

flashBase	The base address of the first flash module (e.g. 0x00400000).
blockMask	A bit mask that defines which of the blocks should be erased.
clearCensor	If set, the censor bits will be erased.At least on block in a module must be selected in order to erase the censor bits of this module.
cpuClock	The CPU clock frequency in Hertz. The BDI needs this parameter to calculate the program/erase pulse timing. It is important to setup the correct frequency otherwise the erase/programming pulses are not correct.
workSpace	This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

2.5.13. BDI_FlashWrite555()

This function programs a S-Record file to the MPC555 internal flash. This is an enhanced versions of the *BDI_FlashWriteFile()* function. It uses a feature of the MPC555 flash that allows to program multiple 64 byte pages at the same time.
Do not use this function when programming shadow information

```
/****************************************************************************
 ****************************************************************************

    Program MPC555 internal Flash memory

    INPUT:  fileName        the full path and filename of the S-record file
    OUTPUT: errorAddr       address of the failing byte
            return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashWrite555(const char *fileName, DWORD *errorAddr);
```

Example:

```
#define MPC555_CLOCK        40000000
#define MPC555_WORKSPACE    0x007FC000

static BDI_InitTypeT mpc555InitList[] = {
  {BDI_INIT_PPC_WMSR,   0,         0x00003002}, // MSR   : set FP,ME,RI
  {BDI_INIT_PPC_WSPR,  27,         0x00003002}, // SRR1  : set FP,ME,RI
  {BDI_INIT_PPC_WSPR, 638,         0x00000802}, // IMMR  : @0x00400000,Flash on
  {BDI_INIT_PPC_WSPR, 158,         0x00000007}, // ICTRL : not serial, no show
  {BDI_INIT_PPC_WM32, 0x006FC004, 0xFFFFFF01}  // SYPCR : disable watchdog
  {BDI_INIT_PPC_WM32, 0x006FC284, 0x0091C000}  // PLPRCR: 40MHz clock
};

static int ProgramMPC555(void)
{
  int   result;
  DWORD errorAddr;

  result = BDI_FlashSetup555(BDI_MPC555_FLASH, MPC555_CLOCK, MPC555_WORKSPACE);
  result = BDI_TargetStartup(0,
                             MPC555_CLOCK,
                             sizeof mpc555InitList / sizeof mpc555InitList[0],
                             mpc555InitList);
  result = BDI_FlashErase555(0x00400000,
                             0xFFFC0000,
                             FALSE,
                             MPC555_CLOCK,
                             MPC555_WORKSPACE);
  result = BDI_FlashWrite555("mpc555.sss", &errorAddr);

  return result;
} /* ProgramMPC555 */
```

2.5.14. BDI_FlashSetupUC3F()

Defines the parameters for MPC56x internal flash programming. Before any other flash related function can be used, this function must be called. The information about the flash type is stored within the access library and is used when programming or erasing flash memory.

```
/****************************************************************************
 ****************************************************************************

    Defines the programming parameters for UC3F flash programming

    INPUT:  flashType       the used Flash Memory Type (25 or 26)
            workSpace       address of workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

/* Flash Type */
#define BDI_UC3F_FLASH      25  /* UC3F (MPC56x) flash array               */
#define BDI_UC3F_SHADOW     26  /* UC3F (MPC56x) flash shadow information  */

int BDI_FlashSetupUC3F(WORD flashType, DWORD workSpace);
```

flashType — Select the flash type:
25 = UC3F internal flash
26 = UC3F internal shadow flash.

workSpace — This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

Example:

```
#define MPC565_CLOCK        20000000
#define MPC565_WORKSPACE    0x003FC000

static BDI_InitTypeT mpc565InitList[] = {
  {BDI_INIT_PPC_WMSR,   0,        0x00003002}, // MSR : set FP,ME,RI
  {BDI_INIT_PPC_WSPR,  27,        0x00003002}, // SRR1: set FP,ME,RI
  {BDI_INIT_PPC_WSPR, 638,        0x00000800}, // IMMR: @0x00000000, Flash on
  {BDI_INIT_PPC_WSPR, 158,        0x00000007}, // ICTRL : not serial, no show
  {BDI_INIT_PPC_WM32, 0x002FC004, 0xFFFFFF01}  // SYPCR : disable watchdog
};

static int ProgramMPC565(void)
{
  int   result;
  DWORD errorAddr;

  result = BDI_FlashSetupUC3F(BDI_UC3F_FLASH, MPC565_WORKSPACE);
  result = BDI_TargetStartup(0,
                             MPC565_CLOCK,
                             sizeof mpc565InitList / sizeof mpc565InitList[0],
                             mpc565InitList);
  result = BDI_FlashEraseUC3F(0x00000000, 0x00F0, FALSE, MPC565_WORKSPACE);
  result = BDI_FlashEraseUC3F(0x00000000, 0x000F, FALSE, MPC565_WORKSPACE);
  result = BDI_FlashEraseUC3F(0x00080000, 0x00F0, FALSE, MPC565_WORKSPACE);
  result = BDI_FlashEraseUC3F(0x00080000, 0x000F, FALSE, MPC565_WORKSPACE);
  result = BDI_FlashWriteFile("mpc565.sss", &errorAddr);

  return result;
} /* ProgramMPC565 */
```

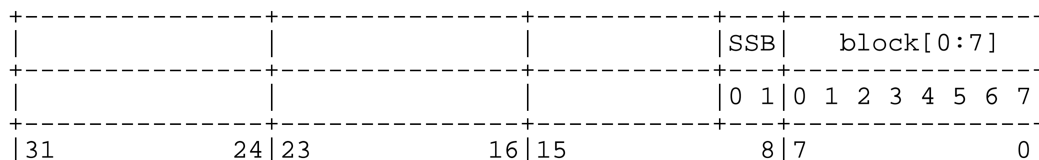
2.5.15. BDI_FlashEraseUC3F()

This function erases one or multiple flash block(s) in the MPC56x internal flash.

```
/****************************************************************************
 ****************************************************************************

    Erase MPC56x internal Flash memory

    The parameter blockMask has the following layout.
```



```
    +---------------+---------------+-----------+---+---------------+
    |               |               |           |SSB|   block[0:7]  |
    +---------------+---------------+-----------+---+---------------+
    |               |               |           |0 1|0 1 2 3 4 5 6 7|
    +---------------+---------------+-----------+---+---------------+
    |31           24|23           16|15        8|7             0|
```

```
    If the appropriate bit is set, the block will be erased.

    INPUT:  flashBase       the base address of the flash module
            blockMask       the mask with the flash block(s) to erase
            clearCensor     if set the censor bits will be erased
            workSpace       address of workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int EXPORT PASCAL FAR BDI_FlashEraseUC3F(DWORD flashBase,
                                         DWORD blockMask,
                                         BOOL  clearCensor,
                                         DWORD workSpace);
```

flashBase	The base address of the first flash module (e.g. 0x00000000).
blockMask	A bit mask that defines which of the blocks should be erased.
clearCensor	If set, the censor bits will be erased.At least on block in a module must be selected in order to erase the censor bits of this module.
workSpace	This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

Note:
Because the DLL timeouts after 5 seconds, erasing the hole flash should be done in multiple steps.

```
// erase complete module A and B within a MPC565
result = BDI_FlashEraseUC3F(0x00000000, 0x00F0, FALSE, MPC565_WORKSPACE);
result = BDI_FlashEraseUC3F(0x00000000, 0x000F, FALSE, MPC565_WORKSPACE);
result = BDI_FlashEraseUC3F(0x00080000, 0x00F0, FALSE, MPC565_WORKSPACE);
result = BDI_FlashEraseUC3F(0x00080000, 0x000F, FALSE, MPC565_WORKSPACE);
```

2.5.16. BDI_FlashSetupSGFM()

Defines the parameters for MMC2114 internal flash programming. Before any other flash related function can be used, this function must be called. The information about the flash is stored within the access library and is used when programming or erasing flash memory.

```
/****************************************************************************
 ****************************************************************************

    Defines the programming parameters for SGFM flash programming

    INPUT:  cpuClock        the target CPU clock in hertz
            workSpace       address of a workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashSetupSGFM(DWORD cpuClock, DWORD workSpace);
```

cpuClock	The CPU clock frequency in Hertz. The BDI needs this parameter to calculate the value for the SGFM Clock Divider register. For proper program and erase operations, it is critical to define the correct CPU clock frequency.
workSpace	This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

Following an example how to erase/program the M-Core MMC2114 Flash module (SGFM) on the MMCEVB2114 evaluation board.

```
#define MMC2114_CLOCK        32000000
#define MMC2114_WORKSPACE    0x00800000

static BDI_InitTypeT mmc2114InitList[] = {
  {BDI_INIT_MMC_WM16,  0x00C70000, 0x000E}, // WCR: disable watchdog
  {BDI_INIT_MMC_WM16,  0x00C30000, 0x6000}, // SYNCR: speed-up clock to 32MHz
  {BDI_INIT_MMC_WM16,  0x00D00000, 0x0000}, // MCR : select block 0
  {BDI_INIT_MMC_WM16,  0x00D00010, 0x0000}, // PROT: unlock all sectors
  {BDI_INIT_MMC_WM16,  0x00D00000, 0x0001}, // MCR : select block 1
  {BDI_INIT_MMC_WM16,  0x00D00010, 0x0000}, // PROT: unlock all sectors
  {BDI_INIT_MMC_WM16,  0x00D00000, 0x0000}, // MCR : select block 0
  {BDI_INIT_MMC_WM8,   0x817FFFFD, 0x10},   // MMIO: enable programming voltage
  {BDI_INIT_MMC_DELAY, 0,          100},    // Delay after enable programming
voltage
};

result = BDI_TargetStartup(0,
                           MMC2114_CLOCK,
                           sizeof mmc2114InitList / sizeof mmc2114InitList[0],
                           mmc2114InitList);
result = BDI_FlashSetupSGFM(MMC2114_CLOCK, MMC2114_WORKSPACE);
result = BDI_FlashErase(BDI_ERASE_BLOCK, 0x00000000);
result = BDI_FlashErase(BDI_ERASE_BLOCK, 0x00020000);
result = BDI_FlashWriteFile("sgfm256k.sss", &errorAddr);
result = BDI_SetByte(0x817FFFFD, 0x00); // MMIO: disable programming voltage
```

2.5.17. BDI_FlashSetupLPC2000()

Defines the parameters for LPC2000 internal flash programming. Before any other flash related function can be used, this function must be called. The information about the flash is stored within the access library and is used when programming or erasing flash memory.

```
/****************************************************************************
 ****************************************************************************

    Defines the programming parameters for LPC2000 flash programming

    INPUT:  flashSize       the flash size
            sysClock        the target system clock in kHz
            workSpace       address of workspace in target RAM
    OUTPUT: return          0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_FlashSetupLPC2000(DWORD flashSize, DWORD sysClock, DWORD workSpace);
```

flashSize	The size of the internal flash. This is 0x20000 (128k) or 0x40000 (256k) or 0x80000 (for all LPC213x and LPC214x devices).
sysClock	The system clock frequency in KHz. The LPC2000 flash driver needs this parameter. For proper program and erase operations, it is critical to define the correct system clock frequency.
workSpace	This value defines the base address of a workspace in target RAM. The workspace is used for a data buffer and to store the algorithm code. There must be at least 2kBytes of RAM available for this purpose.

Following an example how to erase/program the LPC2106 Flash (without error handling):

```
static BDI_InitTypeT initListLPC2106[] = {
  {BDI_INIT_ARM_WGPR, 8006,       100       }, // delay after releasing reset
  {BDI_INIT_ARM_WM32, 0xE01FC040, 0x00000001}, // MEMMAP: User flash mode
};


/* reset and init target */
result = BDI_TargetStartup(500, // 500ms reset time
                           4,   // little endian, ARM7TDMI, 1 MHz JTAG
                           sizeof initListLPC2106 / sizeof initListLPC2106[0],
                           initListLPC2106);

/* setup flash type */
result = BDI_FlashSetupLPC2000(0x20000,      // 128k flash device
                               14745,        // fsys = 14.745MHz
                               0x40001000);  // internal SRAM as workspace

/* erase flash  */
result = BDI_FlashEraseSector(0x00007FFF);  // erase sectors 0 - 14

/* program/verify from a file */
result = BDI_FlashWriteBinary("lpc2000.bin", 0x00000, &errorAddr);
result = BDI_VerifyBinary("lpc2000.bin", 0x00000, &errorAddr);
```

Note: For all LPC213x and LPC214x devices select always 0x80000 (512k) as flashSize. This selects the correct sector map for this devices.

2.6. Miscellaneous Functions

2.6.1. BDI_InfoGet()

This function returns information about the BDI firmware and the current loaded setup. For bdiAccess users, the fields *setupTimeStamp* and *setupId* has no meaning.

```
typedef struct {
  WORD      fwType;
  WORD      fwVersion;
  WORD      cpuType;
  DWORD     setupTimeStamp;
  char      setupId[40];
} BDI_InfoT;

/****************************************************************************
 ****************************************************************************

    Get info about the BDI firmware and the current loaded setup

    INPUT:   -
    OUTPUT: bdiInfo      the BDI info structure
            return       0 if okay or a negativ number if error

 ****************************************************************************/

int BDI_InfoGet(BDI_InfoT * bdiInfo);
```

fwType

The loaded firmware type:

0 = standard firmware for CPU32/CPU16 (HCI,Pro,Spy)
1 = bdiAda firmware for CPU32
2 = bdiWind firmware for CPU32
3 = bdiAda firmware for MPC8xx
4 = bdiWind firmware for MPC8xx
5 = standard firmware for MPC8xx
6 = standard firmware for ColdFire
7 = standard firmware for CPU12
8 = standard firmware for M-CORE
9 = bdiWind firmware for ARM
10 = standard firmware for ARM
11 = <reserved>
12 = bdiGDB firmware for CPU32
13 = bdiGDB firmware for MPC8xx
14 = bdiGDB firmware for ARM
15 = bdiGDB firmware for M-CORE
16 = bdiWind firmware for M-CORE
17 = standard firmware for COP / 603e
18 = bdiWind firmware for COP / 603e
19 = bdiGDB firmware for COP / 603e
20 = standard firmware for PPC4xx
21 = bdiWind firmware for PPC4xx
22 = bdiGDB firmware for PPC4xx
23 = <reserved>
24 = <reserved>
25 = bdiWind firmware for ColdFire
26 = bdiGDB firmware for ColdFire

27 = standard firmware for MPC7450
28 = bdiWind firmware for MPC7450
29 = bdiGDB firmware for MPC7450
30 = standard firmware for MIPS32
31 = bdiWind firmware for MIPS32
32 = bdiGDB firmware for MIPS32
33 = standard firmware for XScale
34 = bdiWind firmware for XScale
35 = bdiGDB firmware for XScale
36 = bdiGDB for MIPS64
37 = standard firmware for MPC85xx
38 = <reserved>
39 = bdiGDB for MPC85xx
40 = standard firmware for ARM11
41 = bdiGDB for ARM11
42 = standard firmware for MIPS64
43 = standard firmware for MPC55xx
44 = bdiGDB for MPC55xx

fwVersion — The version of the loaded firmware:
1..255 = V0.01 ... V2.55

cpuType — The target CPU type (except for HC12 parts, the MSB defines the CPU family and the LSB defines the part within a family) :
0xFFFF = unknown

0x80C0 = CPU12

0x03C0 = HC812A4	0x03C7 = HC912DG128A
0x03C1 = HC912B32	0x03C8 = HC912DT128A
0x03C3 = HC912D60	0x03C9 = MC9S12D256
0x03C4 = HC912DG128	0x03CA = MC9S12D128
0x03C5 = HC912DA128	0x03CB = MC9S12D64
0x03C6 = HC912D60A	0x03CC = MC9S12D32

0x8100 = CPU16
0x8200 = CPU32

0x8300 = ColdFire

0x8301 = MCF5202	0x8302 = MCF5203
0x8303 = MCF5204	0x8304 = MCF5206
0x8305 = MCF5307	0x8306 = MCF5407
0x8307 = MCF5272	0x8308 = ?

0x8400 = MPC8xx

0x8401 = MPC860	0x8402 = MPC821
0x8403 = MPC823	0x8404 = MPC850
0x8405 = MPC801	

0x8500 = MPC5xx

0x8501 = MPC505	0x8502 = MPC509
0x8503 = MPC555	

0x8600 = M-CORE

0x8601 = MMC2001	0x8602 = MMC2107

0x8700 = ARM7TDMI	
0x8701 = ARM7DI	0x8702 = ARM710T
0x8703 = ARM720T	0x8704 = ARM740T
0x8800 = ARM9TDMI	
0x8801 = ARM920T	0x8802 = ARM940T
0x8810 = ARM9E	
0x8811 = ARM946E	0x8812 = ARM966E
0x8880 = XScale	
0x8A00 = PPC603e	
0x8A01 = PPC603ev	0x8A02 = MPC8260
0x8A03 = MPC8240	0x8A04 = MPC4200
0x8B00 = PPC750/740	
0x8C00 = PPC403	
0x8D00 = PPC405	
0x8E00 = PPC7400	
0x8F00 = PPC401	
0x9000 = PPC440	
0x9100 = MIPS32	
0x9101 = RC32300	0x9102 = AU1000
0x9103 = MIPS 4Kc	0x9104 = MIPS 4Km
0x9105 = MIPS 4Kp	

setupTimeStamp	Every time, the setup is written to the flash, a time stamp is also programmed. This time stamp can be used to detect if the newest setup is loaded.
setupId	A zero terminated string with the current loaded setup id.

2.6.2. BDI_UpdateFirmwareLogic()

This function allows to change or update the firmware and/or the logic of the connected BDI. There is no need to use this function because the firmware update will normally done with the bdiAccess configuration software. In special case where the use of the bdiAccess configuration software is not possible, this function can be used.

```
/****************************************************************************
 ****************************************************************************

    BDI_UpdateFirmwareLogic:

     Updates the firmware and the logic of the BDI.

     INPUT  : szPort      a string with the communication parameters
                          (e.g. "COM1 57600")
              szPath      the directory path to the firmware files
              targetType  the connected target type (e.g. BDI_TT_MPC)
              updateMode  the update mode
                            BDI_UPDATE_AUTO
                            BDI_UPDATE_FIRMWARE
                            BDI_UPDATE_LOGIC
                            BDI_UPDATE_ALL
     OUTPUT : RETURN      0 if okay or a negativ number if error.

 ****************************************************************************/

int BDI_UpdateFirmwareLogic(const char FAR *szPort,
                            const char FAR *szPath,
                                  WORD      targetType,
                                  WORD      updateMode);
```

szPort — A string with the communication parameters. Only serial connection is supported for updating the BDI firmware/logic (e.g. "COM1 57600").

szPath — The directory path where the firmware and logic files are stored. The trailing backslash in necessary (e.g. "C:\bdi\setup\").

targetType — This parameter defines the target CPU type for which to load / update the firmware and logic. The appropriate firmware and JEDEC files will be automatically selected. The possible target types are defined in the *bdiifc.h* header file.

updateMode — The update mode defines how update is handled. Normally BDI_UPDATE_AUTO is used. Only in special cases the other modes should be used.

- BDI_UPDATE_AUTO: The firmware/logic is only loaded if it is necessary. It will be loaded if there is a target type change or there are newer versions available in the firmware directory.
- BDI_UPDATE_FIRMWARE: Same as BDI_UPDATE_AUTO but the newest firmware is loaded in any case.
- BDI_UPDATE_LOGIC: Same as BDI_UPDATE_AUTO but the newest logic is loaded in any case.
- BDI_UPDATE_ALL: The newest firmware and logic is loaded in any case, independed if they are already loaded.

After the function has terminated, the connection to the BDI is closed. Also keep in mind, that if an update is necessary, the function needs a long time to complete.

2.6.3. BDI_ConfigNetwork()

This function allows to change or update the network configuration of the connected BDI. There is no need to use this function because the configuration will normally done with the bdiAccess configuration software. In special case where the use of the bdiAccess configuration software is not possible, this function can be used.

```
/****************************************************************************
 ****************************************************************************

    BDI_ConfigNetwork:

     Writes the network parameers to the BDI.

     INPUT  : szPort            a string with the communication parameters
                                (e.g. COM1 57600)
              szBdiIP           the IP address of the BDI
              szSubnetMask      the subnet mask
              szDefaultGateway  the default gateway
     OUTPUT : RETURN            0 if okay or a negativ number if error.

 ****************************************************************************/


int BDI_ConfigNetwork(const char FAR *szPort,
                      const char FAR *szBdiIP,
                      const char FAR *szSubnetMask,
                      const char FAR *szDefaultGateway);
```

szPort	A string with the communication parameters. Only serial connection is supported for updating the network configuration (e.g. "COM1 57600").
SzBdiIP	The IP address of the BDI
SzSubnetMask	The subnet mask or 255.255.255.255 if not used
SzDefaultGateway	The default gateway or 255.255.255.255 if not used

After the function has terminated, the connection to the BDI is closed.

Note:
After using this function the BDI keeps staying in loader mode. A connection via network will fail. To force the BDI out of loader mode, connect to it first via the serial link.

```
iRet = BDI_ConfigNetwork ("COM1 38400",
                          "192.168.0.23",
                          "255.255.255.255",
                          "255.255.255.255");
iRet = BDI_Connect ("COM1 38400");
iRet = BDI_Disconnect();
iRet = BDI_Connect ("NETWORK 192.168.0.23");
```

2.6.4. BDI_GetConfiguation ()

This function allows to read back the BDI configuration. Use it and look what you get.

```
/****************************************************************************
 ****************************************************************************

    BDI_GetConfiguation:

     Read back the BDI configuration.

     INPUT  : szPort             a string with the communication parameters
                                 (e.g. COM1 57600)
     OUTPUT : szBdiType          BDI type an serial number
              szBdiVersions      Loader, Firmware and Logic versions
              szBdiMAC           MAC address
              szBdiIP            IP address
              szSubnetMask       subnet mask
              szDefaultGateway  default gateway
     RETURN   0 if okay or a negativ number if error.

 ****************************************************************************/

int BDI_GetConfiguation(const char *szPort,
                              char *szBdiType,
                              char *szBdiTarget,
                              char *szBdiVersions,
                              char *szBdiMAC,
                              char *szBdiIP,
                              char *szSubnetMask,
                              char *szDefaultGateway);
```

2.6.5. BDI_ErrorTextGet()

Returns a text to a given BDI error number.

```
/****************************************************************************
 ****************************************************************************

    BDI_ErrorTextGet:

     INPUT  : error           the error number
     OUTPUT : RETURN          a pointer to a string with the error text

 ****************************************************************************/

const char* BDI_ErrorTextGet(int error);
```

2.6.6. BDI_ChannelRead(), BDI_ChannelWrite()

Some targets (e.g. ARM7TDMI) support special communication channels between the application running on the target and the debug interface. The following functions allows to communicate with the target application using this special debug communication channels. The special behavior of the used debug communication channel must be observed (e.g. transfer unit is 4 bytes). The functions will time-out when no more data can be written or read. Therefore it is possible that less than the requested count will be transferred.

```
/*************************************************************************
 *************************************************************************

    Read from a debug channel

    INPUT:  channel       the channel ID
            count         number of bytes to read
    OUTPUT: data          the read data will be stored in this buffer
            RETURN        the number of bytes read or a negativ number if error

 *************************************************************************/

int BDI_ChannelRead(WORD channel, WORD count, BYTE FAR *data);
```

```
/*************************************************************************
 *************************************************************************

    Write to a debug channel

    INPUT:  channel       the channel ID
            count         number of bytes to write
            data          data to write to the target memory
    OUTPUT: RETURN        number of bytes written or a negativ number if error

 *************************************************************************/

int BDI_ChannelWrite(WORD channel, WORD count, const BYTE FAR* data)
```

ARM Targets:
For ARM7TDMI targets, the ICEBreaker debug communication channel is accessed with this functions. Set the parameter *channel* to zero for compatibility with future enhancements. Because the transfer unit is 4 bytes, count should be a multiple of 4. The BDI uses little endian byte ordering when converting the byte stream to the 32bit transfer unit and vice versa.

2.6.7. BDI_InstallCallback(), BDI_FileSize()

In order to get an indication about the progress ot to abort of a file download, a callback function can be installed. This callback function is called with the number of already transferred bytes. The return value of this callback function defines if the file downlaod should be aborted.
If installed, this callback function is called during the execution of the following functions:

BDI_LoadFile()

BDI_VerifyFile()

BDI_FlashWriteFile()

```
/****************************************************************************
 ****************************************************************************

    Install a progress callback function

    INPUT:  cb            the callback function
    OUTPUT: -

 ****************************************************************************/

#define BDI_ABORT_REQUEST  0xaba001

typedef int BDI_ProgressCallbackT(DWORD);

void BDI_InstallCallback(BDI_ProgressCallbackT* cb);
```

To get the total number of bytes of a S-record file, the following helper function can be used:

```
int BDI_FileSize(const char *fileName);
```

Example:

```
static int CALLBACK DisplayProgress(DWORD count)
{
  printf("Done %li\r", count);
  return (abortRequest) ? BDI_ABORT_REQUEST : 0;
} /* DisplayProgress */


BDI_Connect();
BDI_InstallCallback(DisplayProgress);
...
BDI_InstallCallback(NULL);
BDI_Disconnect()
```

Note:
Call *BDI_InstallCallback()* after *BDI_Connect()* !

2.6.8. BDI_DoJtag()

For some targets the BDI supports low level JTAG access.

```
/****************************************************************************
 ****************************************************************************

    Low level JTAG access

    INPUT:  command       JTAG command
            control       parameter for the command
            count         number of bits / clocks
            pScanOut      pointer to output scan data
            pScanIn       pointer to buffer for the input scan data
    OUTPUT: RETURN        the number of bytes read or a negativ number if error

 ***************************************************************************/

#define JTAG_CMD_CLOSE          0
#define JTAG_CMD_OPEN           1
#define JTAG_CMD_SPEED          2
#define JTAG_CMD_PORT           3
#define JTAG_CMD_CLOCK          4
#define JTAG_CMD_SCAN           5
#define JTAG_CMD_IRSCAN         6
#define JTAG_CMD_DRSCAN         7

#define JTAG_PORT_MASK          3
#define JTAG_PORT_KEEP          0
#define JTAG_PORT_LOW           1
#define JTAG_PORT_HIGH          2
#define JTAG_PORT_OPENDRAIN     1
#define JTAG_PORT_PUSHPULL      2

#define JTAG_CLOCK_TDO          (1<<1)
#define JTAG_CLOCK_TMS          (1<<0)

#define JTAG_SCAN_LAST          (1<<2)
#define JTAG_SCAN_DATA          (1<<1)
#define JTAG_SCAN_TMS           (1<<0)

#define JTAG_MAX_SCAN           8192


int BDI_DoJtag(       BYTE   command,
                      BYTE   control,
                      WORD   count,
                const BYTE*  pScanOut,
                      BYTE*  pScanIn);
```

command (Byte)	JTAG command 0 = close 4 = clock 1 = open 5 = scan 2 = speed 6 = irscan 3 = port 7 = drscan
control (Byte)	JTAG control option (depends on command)
count (Word)	scan: number of bits to exchange (max. 8 x 1024) clock: number of clocks to apply
pScanOut	pointer to the data to scan out
pScanIn	pointer to buffer or NULL if TDO data should be dropped

Close:
Close the low level JTAG mode

Open:
Open the low level JTAG mode, all other debug commands and Telnet access are blocked.
The control byte defines the port type:

7	6	5	4	3	2	1	0
???		???		RST		TRST	

The two bits in the fields have the following meaning:
- 00 : use default (open-drain or push-pull)
- 01 : select open-drain
- 10 : select push-pull
- 11 : <reserved>

Not all targets support open-drain and push-pull drivers.

Speed:
Sets the JTAG clock frequency, depends on target type, see BDI_TargetStartup () function.

Port:
Allows to set some port signals (at least TRST). What ports are available depends on target type.
The control byte is interpreted as follows:

7	6	5	4	3	2	1	0
???		???		RST		TRST	

The two bits in the fields have the following meaning:
- 00 : let the port as it is
- 01 : force port to low
- 10 : force port to high
- 11 : <reserved>

Clock:
Clock TCK count times (TCK idle is low) with TDO/TMS set as defined in control:

-	-	-	-	-	-	TDO	TMS

Scan:
With this command, all kind of JTAG state machine sequences are possible.
The control parameter has a special meaning.

-	-	-	-	-	LAST	DATA	TMS

TMS: If set then TMS connects to the shifter else TDO (target TDI)
DATA: The data applied to the port that is not connected to the shifter until (count – 1)
LAST: The data applied to the port that is not connected to the shifter for the last TCK

The count parameter defines the number of bits to transfer (number of TCK's).

IRScan, DRScan:
All IR/DR scans can be executed with the above generic Scan command. But to reduce the number of exchanged Command/Answer frames this two commands implement complete IR/DR scans. The TAP has to be in Run-Test/Idle state and will be in Run-Test/Idle state after the scan. Up to 8192 bits can be scanned in the Shift-IR/Shift-DR state.

The byte arrays with the scan data are handled as follows.

```
TDI -> abcdefghijklmnopqrs -> TDO
```

Bit 's' will be scanned first and bit 'a' is the last one scanned.
This bit pattern is placed in the byte array if the following way:

Bit	7	6	5	4	3	2	1	0
pScan[0]	l	m	n	o	p	q	r	s
pScan[1]	d	e	f	g	h	i	j	k
pScan[2]	-	-	-	-	-	a	b	c

Examples:

Move from Run-Test/Idle or Test-Logic Reset to Shift-IR:

```
BDI_DoJtag(JTAG_SCAN, JTAG_SCAN_TMS, 5, {0x06}, NULL);
```

Scan 16 bits in Shift-IR and exit Shift-IR with last data:

```
BDI_DoJtag(JTAG_SCAN, JTAG_SCAN_LAST, 16, data, NULL);
```

Move from Exit1-IR to Run-Test/Idle:

```
BDI_DoJtag(JTAG_SCAN, JTAG_SCAN_TMS, 2, {0x01}, NULL);
```

DR-Scan from Run-Test/Idle to Run-Test/Idle:

```
BDI_DoJtag(JTAG_DRSCAN, 0, count, dataOut, dataIn);
```

IR-Scan from Run-Test/Idle to Run-Test/Idle, drop target TDO:

```
BDI_DoJtag(JTAG_IRSCAN, 0, count, dataOut, NULL);
```

2.7. Additional information

2.7.1. Flash Programming for HC12

This chapter gives some additional information how to program internal and external flash memories on HC12 targets.

2.7.1.1. Internal EEPROM

The following example shows how to erase and program the internal EEPROM of a HC12 target. For ease of understanding, error handling is not shown.

IMPORTANT: For HC12A EEPROM, do not forget to setup EEDIVH / EEDIVL
IMPORTANT: For Star12 EEPROM, do not forget to setup ECLKDIV

MC68HC812A4:

First, connect to the BDI:

```
result = BDI_Connect("COM1 57600");
```

Define the memory type to erase or program. This step should be done immediately after connecting to the BDI because this will setup the BDI firmware for flash programming.
SRAM is assumed to be at the default position of 0x800.

```
result = BDI_FlashSetupHC12(BDI_CPU_HC812A4, BDI_HC12_EEPROM, 8, 4096, 0x800);
```

Reset the target and setup some registers:

```
result = BDI_SetBdmSpeed(6); /* 8MHz for BDI1000 */
result = BDI_TargetReset();
result = BDI_SetByte(0x0012, 0x11); /* INITEE: Map EEPROM to 0x1000 */
result = BDI_SetByte(0x00F1, 0x00); /* EEPROT: Disable EEPROM Protection */
```

This erases the internal EEPROM. The EEPROM is mapped to address 0x1000 and the size is 4096 bytes:

```
result = BDI_FlashEraseHC12(0x1000, 4096);
```

Program and verify the EEPROM.

```
result = BDI_FlashWriteBlock(0x1000, 4096, dataBuffer, &errorAddr);
result = BDI_VerifyBlock(0x1000, 4096, dataBuffer, &errorAddr);
```

It's also possible to progam the content of a S-record file into the EEPROM:

```
result = BDI_FlashWriteFile("eeprom.sss", &errorAddr);
result = BDI_VerifyFile("eeprom.sss", &errorAddr);
```

MC69HC912B32:

The following example programs the EEPROM of a MC69HC912B32. The 768 bytes EEPROM are mapped to the address 0x3D00:

```
result = BDI_FlashSetupHC12(BDI_CPU_HC912B32, BDI_HC12_EEPROM, 8, 768, 0x800);
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0012, 0x31); /* INITEE: Map EEPROM to 0x3D00 */
result = BDI_SetByte(0x00F1, 0x00); /* EEPROT: Disable EEPROM Protection */
result = BDI_FlashEraseHC12(0x3D00, 768);
result = BDI_FlashWriteBlock(0x3D00, 768, dataBuffer, &errorAddr);
```

MC9S12DP256:

The following example programs the EEPROM of a MC9S12DP256. The 4096 bytes EEPROM are mapped to the address 0x3000:

```
result = BDI_FlashSetupHC12(  BDI_CPU_MC9S12D256,
                              BDI_STAR12_EEPROM,
                              8,
                              4096,
                              0x800);
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0012, 0x31); /* INITEE:  Map EEPROM to 0x3000 */
result = BDI_SetByte(0x0010, 0x00); /* INITRM:  Map RAM to 0x0000 */
result = BDI_SetByte(0x0110, 0x49); /* ECLKDIV: Clock divider for 16MHz  */
result = BDI_SetByte(0x0114, 0xFF); /* EPROT:  disable EEPROM protection */
result = BDI_FlashEraseHC12(0x3000, 4096);
result = BDI_FlashWriteBlock(0x3000, 0xFF0, dataBuffer, &errorAddr);
result = BDI_VerifyBlock(0x3000, 0xFF0, dataBuffer, &errorAddr);
```

MC9S12DPX512:

The following example programs the EEPROM of a MC9S12XDP512.
The addresses in the file have to be global (or paged) ones.
The global address range for this EEPROM is from 0x13F000 to 0x13FFFF.

```
result = BDI_FlashSetupHC12(BDI_CPU_S12X_512K4,
                            BDI_STAR12_EEPROM (| BDI_HC12_PAGED),
                            8,
                            4096,
                            0x2000);
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0110, 0x49); /* ECLKDIV: Clock divider for 16MHz */
result = BDI_SetByte(0x0114, 0xFF); /* EPROT: disable EEPROM protection */
result = BDI_FlashEraseHC12(0x0800, 4096);
result = BDI_FlashWriteFile("D:/hc12/s12x_eetx4k.sss", &errorAddr);
result = BDI_VerifyFile("D:/hc12/s12x_eetx4k.sss", &errorAddr);
```

2.7.1.2. Internal Flash

The following example shows how to erase and program the internal flash of a HC12 target.
For ease of understanding, error handling is not shown.

IMPORTANT: For Star12 flash, do not forget to setup FCLKDIV

MC68HC912B32:

First, connect to the BDI:

```
result = BDI_Connect("COM1 57600");
```

Define the memory type to erase or program. This step should be done immediately after connecting to the BDI because this will setup the BDI firmware for flash programming.
SRAM is assumed to be at the default position of 0x800.
For internal flash, bus width must be set to 16 bit.

```
result = BDI_FlashSetupHC12(   BDI_CPU_HC912B32,
                               BDI_HC12_FLASH,
                               16,
                               0x8000,
                               0x800);
```

Reset the target and setup some registers:

```
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0012, 0x11); /* INITEE: Map EEPROM to 0x1D00 */
result = BDI_SetByte(0x00F5, 0x00); /* FEEMCR: Enable program of boot block */
```

Erase the internal flash

```
result = BDI_FlashEraseHC12(0x8000, 0x8000);
```

Program the flash with data from a S-record file. The S-records should be word aligned.

```
result = BDI_FlashWriteFile("flash32.sss", &errorAddr);
```

If you wish you can also verify the programmed data

```
result = BDI_VerifyFile("flash32.sss", &errorAddr);
```

MC69HC912D60:

The following example erases and programs the internal flash of a MC69HC912D60:

```
result = BDI_FlashSetupHC12(BDI_CPU_HC912D60,BDI_HC12_FLASH,16,0x1000,0x800);
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0010, 0x08); /* INITRM: Map RAM to 0x0800 */
result = BDI_SetByte(0x00F5, 0x00); /* FEE32MCR: Enable boot block */
result = BDI_SetByte(0x00F9, 0x00); /* FEE28MCR: Enable boot block */
result = BDI_FlashEraseHC12(0x1000, 0x7000); /* erase 28k flash block */
result = BDI_FlashEraseHC12(0x8000, 0x8000); /* erase 32k flash block */
result = BDI_FlashWriteFile("flash60.sss", &errorAddr);
result = BDI_VerifyFile("flash60.sss", &errorAddr);
```

MC69HC912DG128A:

The following example erases and programs the internal flash of a MC69HC912DG128A.

```
result = BDI_FlashSetupHC12(  BDI_CPU_HC912DG128A,
                              BDI_HC12A_FLASH,
                              16,
                              0x20000,
                              0x800);
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x000B, 0x00); /* MODE: Disable Internal Visibility */
result = BDI_SetByte(0x000B, 0x00); /* MODE: Write twice, first is ignored */
result = BDI_SetByte(0x0010, 0x00); /* INITRM: Map RAM to 0x0000  */
result = BDI_SetByte(0x00FF, 0x00); /* PPAGE: Select Flash Array 0  */
result = BDI_SetByte(0x00F5, 0x00); /* FEEMCR: Enable program of boot block  */
result = BDI_SetByte(0x00FF, 0x02); /* PPAGE: Select Flash Array 1  */
result = BDI_SetByte(0x00F5, 0x00); /* FEEMCR: Enable program of boot block  */
result = BDI_SetByte(0x00FF, 0x04); /* PPAGE: Select Flash Array 2  */
result = BDI_SetByte(0x00F5, 0x00); /* FEEMCR: Enable program of boot block  */
result = BDI_SetByte(0x00FF, 0x06); /* PPAGE: Select Flash Array 3  */
result = BDI_SetByte(0x00F5, 0x00); /* FEEMCR: Enable program of boot block  */
result = BDI_FlashEraseHC12(0x08000, 0x8000); /* erase 32k flash block 0 */
result = BDI_FlashEraseHC12(0x28000, 0x8000); /* erase 32k flash block 1 */
result = BDI_FlashEraseHC12(0x48000, 0x8000); /* erase 32k flash block 2 */
result = BDI_FlashEraseHC12(0x68000, 0x8000); /* erase 32k flash block 3 */
result = BDI_FlashWriteFile("flash128.sss", &errorAddr);
result = BDI_VerifyFile("flash128.sss", &errorAddr);
```

Important note:
Normally the addresses in the S-record file are interpreted as linear addresses going from 0x00000000 to 0x00020000 for a 128 kB flash memory. The BDI uses always the program page window to access the flash memory.

S.record Address	Page (PPAGE)	HC12 internal address
0x00000000	0x00	0x8000
0x00003FFF	0x00	0xBFFF
0x00004000	0x01	0x8000
...	...	...
0x0001C000	0x07	0x8000
0x0001FFFF	0x07	0xBFFF

If the S-record file to program uses paged addresses with the following structure, add the paged bit to the "flashType" parameter of the "BDI_FlashSetupHC12()" function.

```
result = BDI_FlashSetupHC12(  BDI_CPU_HC912DG128A,
                              BDI_HC12A_FLASH | BDI_HC12_PAGED,
                              16,
                              0x20000,
                              0x800);
```

With this bit set, the DLL translates the addresses first to linear ones before it sends a data block to the BDI. Blocks with an address below 0x4000 (e.g. EEPROM data) are simply ignored.

MC9S12DP256:

The following example erases and programs the internal flash of a MC9S12DP256.

```
result = BDI_FlashSetupHC12(BDI_CPU_MC9S12D256,
                            BDI_STAR12_FLASH | BDI_HC12_PAGED,
                            16,
                            0x40000,
                            0x800);

result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0010, 0x00); /* INITRM: Map RAM to 0x0000 */
result = BDI_SetByte(0x0100, 0x49); /* FCLKDIV: Clock divider for 16MHz */
result = BDI_SetByte(0x0103, 0x00); /* FCNF: select flash block 0  */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */
result = BDI_SetByte(0x0103, 0x01); /* FCNF: select flash block 1  */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */
result = BDI_SetByte(0x0103, 0x02); /* FCNF: select flash block 2  */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */
result = BDI_SetByte(0x0103, 0x03); /* FCNF: select flash block 3  */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */
result = BDI_SetByte(0x0103, 0x00); /* FCNF: select flash block 0  */

/* erase flash  */
result = BDI_FlashEraseHC12(0x3C8000, 0x10000); /* erase 64k flash block 0 */
result = BDI_FlashEraseHC12(0x388000, 0x10000); /* erase 64k flash block 1 */
result = BDI_FlashEraseHC12(0x348000, 0x10000); /* erase 64k flash block 2 */
result = BDI_FlashEraseHC12(0x308000, 0x10000); /* erase 64k flash block 3 */

/* program from a file */
result = BDI_FlashWriteFile("paged256.sss", &errorAddr);
result = BDI_VerifyFile("paged256.sss", &errorAddr);
```

MC9S12XDP512:

The following example erases and programs the internal flash of a MC9S12XDP512.
The addresses in the file have to be global (or paged) ones.
The global address range for this flash is from 0x780000 to 0x7FFFFF.

```
result = BDI_FlashSetupHC12(BDI_CPU_S12X_512K4,
                            BDI_STAR12_FLASH (| BDI_HC12_PAGED),
                            16,
                            0x80000,
                            0x2000);

result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0100, 0x49); /* FCLKDIV: clock divider for 16MHz */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */

/* erase flash  */
result = BDI_FlashEraseHC12(0xE08000, 0x20000);
result = BDI_FlashEraseHC12(0xE88000, 0x20000);
result = BDI_FlashEraseHC12(0xF08000, 0x20000);
result = BDI_FlashEraseHC12(0xF88000, 0x20000);

/* program from a file */
result = BDI_FlashWriteFile("D:/hc12/s12x_ftx512k4.sss", &errorAddr);
result = BDI_VerifyFile("D:/hc12/s12x_ftx512k4.sss", &errorAddr);
```

How to erase the flash of the different HCS12 devices:

32K:
```
result = BDI_FlashEraseHC12(0x3E8000, 0x8000);
```

64K:
```
result = BDI_FlashEraseHC12(0x3C8000, 0x10000);
```

128K (2 x 64K):
```
result = BDI_FlashEraseHC12(0x388000, 0x10000);
result = BDI_FlashEraseHC12(0x3C8000, 0x10000);
```

256K (4 x 64K):
```
result = BDI_FlashEraseHC12(0x308000, 0x10000);
result = BDI_FlashEraseHC12(0x348000, 0x10000);
result = BDI_FlashEraseHC12(0x388000, 0x10000);
result = BDI_FlashEraseHC12(0x3C8000, 0x10000);
```

512K (4 x 128K):
```
result = BDI_FlashEraseHC12(0x208000, 0x20000);
result = BDI_FlashEraseHC12(0x288000, 0x20000);
result = BDI_FlashEraseHC12(0x308000, 0x20000);
result = BDI_FlashEraseHC12(0x388000, 0x20000);
```

256K (2 x 128K):
```
result = BDI_FlashEraseHC12(0x308000, 0x20000);
result = BDI_FlashEraseHC12(0x388000, 0x20000);
```

128K (1 x 128K):
```
result = BDI_FlashEraseHC12(0x388000, 0x20000);
```

How to erase the flash of the different S12X devices:

128K (1 x 128K):
```
result = BDI_FlashEraseHC12(0xF88000, 0x20000);
```

256K (2 x 128K):
```
result = BDI_FlashEraseHC12(0xF08000, 0x20000);
result = BDI_FlashEraseHC12(0xF88000, 0x20000);
```

512K (4 x 128K):
```
result = BDI_FlashEraseHC12(0xE08000, 0x20000);
result = BDI_FlashEraseHC12(0xE88000, 0x20000);
result = BDI_FlashEraseHC12(0xF08000, 0x20000);
result = BDI_FlashEraseHC12(0xF88000, 0x20000);
```

2.7.1.3. External Flash

For the HC812A4, bdiAccess supports programming of external flash devices. Up to 4 Mbyte of external flash can be handled. The BDI uses always the program page window to access the external flash device. The program page register (PPAGE) is automatically set according to the accessed address. Write to the appropriate register to enable the program window and the used expansion address pins (A16...?).

Because of speed, the programming algorithm runs in target RAM. Therefore the BDI needs some RAM space in the target memory map. This RAM is used to store the programming algorithm and the data to program. The BDI assumes a standard memory map with the registers at address 0x0000 and at least 1 Kbyte RAM starting at 0x0800. Write to the appropriate register to map the RAM into this range.

During programming, the addresses are interpreted as physical flash addresses. This is the address that is present at the address pins of the HC812A4.

Flash Address	Page (PPAGE)	HC12 internal address
0x00000000	0x00	0x8000
0x00003FFF	0x00	0xBFFF
0x00004000	0x01	0x8000
...	...	...
0x003FC000	0xFF	0x8000
0x003FCFFFF	0xFF	0xBFFF

The following bdiAccess functions will automatically switch to this address translation mode:

```
BDI_DumpBlock()         BDI_DumpFile()
BDI_VerifyBlock()       BDI_VerifyFile()
BDI_FlashWriteBlock()   BDI_FlashWriteFile()
```

Note:
To switch back to normal addressing mode (e.g. in order to access the register block) the reset function *BDI_TargetReset()* can be used.

Example:
The following example shows how to erase and program AMD29F010 devices. There are two AMD29F010 devices used to build a 16bit flash memory system.

First, connect to the BDI:

```
result = BDI_Connect("COM1 57600");
```

Define the memory type to erase or program. This step should be done immediately after connecting to the BDI because this will setup the BDI firmware for flash programming.
SRAM is assumed to be at the default position of 0x800. There is a 16bit memory bus and the size of **one** AMD29F010 device is 128kb (0x20000).

```
result = BDI_FlashSetupHC12(  BDI_CPU_HC812A4,
                              BDI_FLASH_AM29F0X0,
                              16,
                              0x20000,
                              0x800);
```

Reset the target and setup the paging registers:

```
result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x000B, 0x6B); /* MODE: Special Exp. Wide Mode */
result = BDI_SetByte(0x000B, 0x6B); /* MODE: Write twice, first is ignored */
result = BDI_SetByte(0x0038, 0x03);  /* MXAR: Enable A16E A17E (256kByte) */
result = BDI_SetByte(0x0037, 0x40); /* WINDEF: enable program window */
```

Erase all flash 8 sectors. The two chips are always accessed at the same time.

```
result = BDI_FlashEraseHC12(0x00000, 0);
result = BDI_FlashEraseHC12(0x08000, 0);
result = BDI_FlashEraseHC12(0x10000, 0);
result = BDI_FlashEraseHC12(0x18000, 0);
result = BDI_FlashEraseHC12(0x20000, 0);
result = BDI_FlashEraseHC12(0x28000, 0);
result = BDI_FlashEraseHC12(0x30000, 0);
result = BDI_FlashEraseHC12(0x38000, 0);
```

Now we program and verify some part of the flash memory.

```
result = BDI_FlashWriteBlock(0x08000, 4096, dataBuffer, &errorAddr);
result = BDI_FlashWriteBlock(0x18000, 4096, dataBuffer, &errorAddr);
result = BDI_VerifyBlock(0x08000, 4096, dataBuffer, &errorAddr);
result = BDI_VerifyBlock(0x18000, 4096, dataBuffer, &errorAddr);
```

We can also program a S-record file into the flash

```
result = BDI_FlashWriteFile("extflash.sss", &errorAddr);
result = BDI_VerifyFile("extflash.sss", &errorAddr);
```

2.7.1.4. Working in debug mode

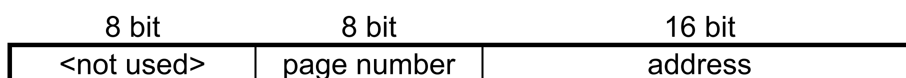
If the bdiAccess library should be used for other tasks than flash programming, e.g. downloading and executing a program, it's best to keep the BDI firmware in debug mode. After power-up of the BDI it is in debug mode until a call to the function *BDI_FlashSetupHC12()* is made. In debug mode, the following things are different:

- To reset / init the target, the function *BDI_TargetStartup()* must be used.
- The initialization list used for the *BDI_TargetStartup()* function must setup the final memory map. This because the BDI reads back information from the target register file after processing this initialization list. For examples, the BDI knows where the EEPROM is mapped and uses automatically the appropriate programming algorithm.
- For a HC812A4 device, the BDI detects if paging is active for external memory and handles the page register automatically.
- Address translation for paged external memory is different than in programming mode.
- The normal memory write functions can be used to write to the internal EEPROM.

Paging in debug mode:

For HC12 targets the BDI handles Code and Data paging automatically (e.g. for the HC812A4). The debugger uses always logical (24bit) addresses. The BDI automatically writes to the appropriate page register before accessing the target memory.

The BDI assumes the following structure of the address:



address	The value in this field must point into the appropriate page window (e.g. 0x8000 to 0xBFFF for program page addresses). Based on this part of the address, the BDI can select the proper page window (Code, Data or Extra window).
page number	The page number that will be written to the appropriate page register.
<not used>	The upper 8 bits of the 32 bit address value should be zero.

After the BDI has processed the initialization list, it reads back information about the paging (enabled or disabled, position of Extra window). Because of this behavior, the initialization list must setup the final memory map.

Example for a HC812A4 target:

```
static BDI_InitTypeT initList[] = {
  {BDI_INIT_CPU12_WM8, 0x0012, 0x11 }, // INITEE: Map EEPROM to 0x1000
  {BDI_INIT_CPU12_WM8, 0x00F1, 0x00 }, // EEPROT: Unprotect all EEPROM blocks
  {BDI_INIT_CPU12_WM8, 0x000B, 0x6B }, // MODE: Special Exp. Wide Mode
  {BDI_INIT_CPU12_WM8, 0x000B, 0x6B }, // MODE: Write twice, first is ignored
  {BDI_INIT_CPU12_WM8, 0x0038, 0x03 }, // MXAR: Enable A16E A17E (256kByte SRAM)
  {BDI_INIT_CPU12_WM8, 0x0037, 0x40 }  // WINDEF: enable program window
};

/* reset and init target */
result = BDI_TargetStartup(100,
                           0x08000001, // SRAM=0x0800, REG=0x0000, HC812A4, 8MHz
                           sizeof initList / sizeof initList[0],
                           initList);
```

2.7.1.5. Unsecuring a HCS12 device

Newer HCS12 devices (e.g. HC9S12DP256 mask set 1K79X) support unsecuring the chip via BDM. The latest BDI firmware supports this feature. With the following call to *BDI_TargetStartup()* you can trigger this unsecuring sequence. This call has no effect if the device is not secured.

```
static BDI_InitTypeT initUnsecure[] = {
  {BDI_INIT_CPU12_WM8, 0x0110, 0x49 }, /* ECLKDIV: EEPROM divider for 16MHz */
  {BDI_INIT_CPU12_WM8, 0x0100, 0x49 }  /* FCLKDIV: Flash  divider for 16MHz */
};

/* trigger unsecure sequence */
result = BDI_TargetStartup(100,
                           (BDI_CPU_MC9S12D256 << 8) | 6, /* D256, BDM 8MHz */
                           sizeof initUnsecure / sizeof initUnsecure[0],
                           initUnsecure);
```

In the case that the device is secured but flash and EEPROM are erased, the above sequence has also no effect. To bring the device in the unsecured state, erase block 0 of the flash:

```
result = BDI_FlashSetupHC12(BDI_CPU_MC9S12D256,
                            BDI_STAR12_FLASH,
                            16,
                            0x40000,
                            0x800);

result = BDI_SetBdmSpeed(6);
result = BDI_TargetReset();
result = BDI_SetByte(0x0010, 0x00); /* INITRM: Map RAM to 0x0000 */
result = BDI_SetByte(0x0100, 0x49); /* FCLKDIV: Clock divider for 16MHz */
result = BDI_SetByte(0x0103, 0x00); /* FCNF: select flash block 0  */
result = BDI_SetByte(0x0104, 0xFF); /* FPROT: disable flash protection  */

result = BDI_FlashEraseHC12(0x3C8000, 0x10000); /* erase 64k flash block 0 */
```

2.7.2. Programming the ColdFire Flash Module (CFM)

To erase and program the ColdFire Flash Module (CFM) you have to access it via the backdoor address (IPSBAR + 0x04000000). This backdoor address has to be used for erase and program commands. Before you can erase/program the CFM, the CFM Clock Divider needs to be setup via an init list entry. Check the MCF5282 user's manual about how to setup the CFMCLKD.

WARNING:
For proper program and erase operations, it is critical to set f_{CLK} between 150 kHz and 200 kHz. Array damage due to overstress can occur when f_{CLK} is less than 150 kHz. Incomplete programming and erasure can occur when f_{CLK} is greater than 200 kHz.

The following example shows how to erase/program the the whole flash of aMCF5282. Please notice the use of 0x44000000 as flash base address even the flash is mapped to 0xf0000000. Also the S-Record file you would like to program has to be mapped to 0x44000000.

```
#define MCF5282_CLOCK        64000000
#define MCF5282_WORKSPACE    0x20000000

static BDI_InitTypeT mcf5282InitList[] = {
{BDI_INIT_MCF_WCREG, 0xC05,      0x20000001}, // RAMBAR: SRAM at 0x20000000
{BDI_INIT_MCF_WM16,  0x40140000, 0x000E    }, // WCR   : Disable watchdog
{BDI_INIT_MCF_WM16,  0x40120000, 0x2000    }, // SYNCR : Speed-up PLL to 64MHz
{BDI_INIT_MCF_DELAY, 0,          10        }, // Delay after changing the PLL
{BDI_INIT_MCF_WCREG, 0xC04,      0xF0000121}, // FLASHBAR: Flash at 0xf0000000
{BDI_INIT_MCF_WM8,   0x401D0002, 0x54      }, // CFMCLKD: Clk divider for 64MHz
{BDI_INIT_MCF_WM32,  0x401D0010, 0x00000000}, // CFMPROT: disable protection
};

result = BDI_TargetStartup(0,
                           MCF5282_CLOCK,
                           sizeof mcf5282InitList / sizeof mcf5282InitList[0],
                           mcf5282InitList);
result = BDI_FlashSetType(BDI_FLASH_CFM, 0x40000, 0, 32, MCF5282_WORKSPACE);
result = BDI_FlashErase(BDI_ERASE_BLOCK, 0x44000000);
result = BDI_FlashErase(BDI_ERASE_BLOCK, 0x44040000);
result = BDI_FlashWriteFile("m5282cfm.sss", &errorAddr);
```

It is also possible to erase only one page:

```
result = BDI_FlashErase(BDI_ERASE_SECTOR, 0x44040800);
```

2.7.3. Programming the MAC7100 Flash Module (CFM32, CFM16)

To erase and program the MAC7100 Program Flash Module (CFM32) you have to access it via the programming interface address (0xFC100000). This programming interface address has to be used for erase and program commands. Before you can erase/program the Program or Data Flash module, the CFM Clock Divider needs to be setup via an init list entry. Check the MAC7100 user's manual about how to setup the CFMCLKD. Keep in mind that the input frequency for the flash module is the IP clock with a frequency of ½ fsys. If for example the system clock is 8 MHz, the clock input to the flash is 4MHz and the correct value for CFMCKLD is 19 (0x13).

WARNING:
For proper program and erase operations, it is critical to set f_{CLK} between 150 kHz and 200 kHz. Array damage due to overstress can occur when f_{CLK} is less than 150 kHz. Incomplete programming and erasure can occur when f_{CLK} is greater than 200 kHz.

The following example shows how to erase/program the the whole program flash of a MAC7111. Please notice the use of 0xFC100000 as flash base address even the flash is mapped to 0x00000000 or 0x20000000. The system frequency is 8MHz (default on MAC7100EVB).
Also the S-Record file you would like to program has to be mapped to 0xFC100000.

```
#define MAC7100_WORKSPACE    0x40000000

static BDI_InitTypeT mac7100InitList[] = {
{BDI_INIT_ARM_GPR,  8008,       0x40000000}, // Save debug PC in internal SRAM
{BDI_INIT_ARM_WM32, 0xFC0F0010, 0x00000000}, // CFMPROT  : disable protection
{BDI_INIT_ARM_WM8,  0xFC0F0044, 0x00      }, // CFMDFPROT: disable potection
{BDI_INIT_ARM_WM8,  0xFC0F0002, 0x13      }, // CFMCLKD  : 4.0 MHz IP clock
};


result = BDI_TargetStartup(
         100,                       // 100ms reset time
         (1 << 16) | (13 << 8) | 5, // big endian, MAC7100, 500kHz JTAG clock
         sizeof mac7100InitList / sizeof mac7100InitList [0],
         mac7100InitList);
result = BDI_FlashSetType(BDI_FLASH_CFM32, 0x80000, 0, 32, MAC7100_WORKSPACE);
result = BDI_FlashErase(BDI_ERASE_BLOCK, 0xFC100000); // mass erase
result = BDI_FlashWriteFile("mac7100cfm.sss", &errorAddr);
```

It is also possible to erase only 4k pages:

```
result = BDI_FlashErase(BDI_ERASE_SECTOR, 0xFC101000);
result = BDI_FlashErase(BDI_ERASE_SECTOR, 0xFC102000);
```

2.7.4. Programming the ADuC7020 Flash Memory

The following example shows how to erase and program the internal flash of the ADuC7020.

```
#define ADUC7020_WORKSPACE    0x00010020

static BDI_InitTypeT initListADUC7020[] = {
  {BDI_INIT_ARM_WGPR,  8006, 300       }, // delay after releasing reset
  {BDI_INIT_ARM_WGPR,  8008, 0x00010000}, // Set save debug PC to internal SRAM
  {BDI_INIT_ARM_WCPSR, 0,    0x000000D3}, // CPSR: set superviser mode
};


/* reset and init target */
result = BDI_TargetStartup(100, // 100ms reset time
                           1,   // little endian, ARM7TDMI, 16 MHz JTAG
                           sizeof initListADUC7020/sizeof initListADUC7020 [0],
                           initListADUC7020);

/* setup flash type */
result = BDI_FlashSetType(BDI_FLASH_ADUC7000,
                          0x10000,
                          0,
                          16,
                          ADUC7020_WORKSPACE);


/* erase flash sector by sector */
for (sector = 0x80000; sector <= 0x8F600; sector += 0x200) {
  result = BDI_FlashEraseSector(sector);
}

OR

/* erase whole chip */
result = BDI_FlashErase(BDI_ERASE_CHIP, 0x80000);

/* program/verify from a file */
result = BDI_FlashWriteFile("aduc7020.sss", &errorAddr);
result = BDI_VerifyFile("aduc7020.sss", &errorAddr);
```

2.7.5. Programming the ST STA2051 Flash Memory

The following example shows how to erase and program the internal flash of the STA2051.
The second BDI_TargetStartup() maybe necessary when the code stored previously n the flash puts the devise in a state where programming with target algorithm fails.

```
#define STA2051_WORKSPACE    0x20000000

static BDI_InitTypeT initListSTA2051[] = {
  {BDI_INIT_ARM_WGPR,  8006, 300       }, // delay after releasing reset
  {BDI_INIT_ARM_WGPR,  8008, 0x20000000}, // Set save debug PC to internal SRAM
  {BDI_INIT_ARM_WGPR,  15,   0x00000000}, // Set PC to 0x00000000
  {BDI_INIT_ARM_WCPSR, 0,    0x000000D3}, // CPSR: set superviser mode
};


/* reset and init target */
result = BDI_TargetStartup(500, // 500ms reset time
                           2,   // little endian, ARM7TDMI, 8 MHz JTAG
                           sizeof initListSTA2051 / sizeof initListSTA2051[0],
                           initListSTA2051);

  /* setup flash type */
result = BDI_FlashSetType(BDI_FLASH_STA2051,
                          0x20000,
                          0,
                          32,
                          STA2051_WORKSPACE);

/* erase flash  */
result = BDI_FlashEraseSector(0x0000001f); // erase B0F0 - B0F4
result = BDI_FlashEraseSector(0x00000020); // erase B0F5
result = BDI_FlashEraseSector(0x00000040); // erase B0F6
result = BDI_FlashEraseSector(0x00000080); // erase B0F7
result = BDI_FlashEraseSector(0x00030000); // erase B1F0 - B1F1

/* reset target again if old flash code causes programming error */
result = BDI_TargetStartup(500, // 500ms reset time
                           2,   // little endian, ARM7TDMI, 8 MHz JTAG
                           sizeof initListSTA2051 / sizeof initListSTA2051[0],
                           initListSTA2051);

/* program/verify from a file */
result = BDI_FlashWriteFile("sta2051_256k.sss", &errorAddr);
result = BDI_VerifyFile("sta2051_256k.sss", &errorAddr);
```

2.7.6. Programming the ST ST30F774 Flash Memory

The following example shows how to erase and program the internal flash of the ST30F774.

```
#define ST30F_WORKSPACE    0xA0000000

static BDI_InitTypeT initListST30F774 [] = {
  {BDI_INIT_ARM_WGPR,  8009, 1         }, // TRST driver type is push-pull
  {BDI_INIT_ARM_WGPR,  8007, 2         }, // Special ST30 reset sequence
  {BDI_INIT_ARM_WGPR,  8006, 1000      }, // Delay for 1000 us after TRST/RST
  {BDI_INIT_ARM_WGPR,  8008, 0xA0000000}, // Set save debug PC to internal SRAM
  {BDI_INIT_ARM_WGPR,  15,   0x00000000}, // Set PC to 0x00000000
  {BDI_INIT_ARM_WCPSR, 0,    0x000000D3}, // CPSR: set superviser mode
};



/* reset and init target */
result = BDI_TargetStartup(100, // 100ms reset time
                           2,   // little endian, ARM7TDMI, 8 MHz JTAG
                           sizeof initListST30F774/ sizeof initListST30F774 [0],
                           initListST30F774);

  /* setup flash type */
result = BDI_FlashSetType(BDI_FLASH_ST30F,
                          0x80000,
                          0,
                          32,
                          ST30F_WORKSPACE);

/* erase flash  */
result = BDI_FlashEraseSector(0x00000fff); // erase B0F0 - B0F11

/* program/verify from a file */
result = BDI_FlashWriteFile("st30f_512k.sss", &errorAddr);
result = BDI_VerifyFile("st30f_512k.sss", &errorAddr);
```

2.7.7. Programming the Atmel AT91SAM7Sxx Flash Memory

The BDI supports programming of the Atmel AT91SAM7S internal flash. Before using any flash function it is important that the MC_FMR is programmed with the correct values for FMCN and FWS. This can be done via the initialization list.

The BDI supports erasing a page or the complete flash memory.

The following example shows how to erase and program the internal flash of the AT91SAM7S64.

```
static BDI_InitTypeT initListAT91SAM7S[] = {
  {BDI_INIT_ARM_WM32,   0xFFFFFD44, 0x00008000}, // Disable watchdog
  {BDI_INIT_ARM_WM32,   0xFFFFFD08, 0xA5000001}, // Enable user reset
  {BDI_INIT_ARM_WM32,   0xFFFFFC20, 0x00000601}, // CKGR_MOR: Enable Main Osci
  {BDI_INIT_ARM_DELAY,  0,          20        },
  {BDI_INIT_ARM_WM32,   0xFFFFFC2C, 0x10480A0E}, // CKGR_PLLR: 96.1MHz
  {BDI_INIT_ARM_DELAY,  0,          20        },
  {BDI_INIT_ARM_WM32,   0xFFFFFC30, 0x00000007}, // PMC_MCKR: MCK = PLL/2=48MHz
  {BDI_INIT_ARM_DELAY,  0,          20        },
  {BDI_INIT_ARM_WGPR,   8005,       0x00000001}, // Set new JTAG clock to 16 MHz
  {BDI_INIT_ARM_WM32,   0xFFFFFF60, 0x00300100}, // MC_FMR: FWS=1,FMCN=48
};


/* reset and init target */
result = BDI_TargetStartup(100, // 100ms reset time
                           10,  // little endian, ARM7TDMI, 10kHz JTAG
                           sizeof initListAT91SAM7S/sizeof initListAT91SAM7S[0],
                           initListAT91SAM7S);

/* setup flash type */
result = BDI_FlashSetType(BDI_FLASH_AT91SAM7S,  // AT91SAM7S64K
                          0x10000,              // 64K
                          0,
                          32,
                          0xFFFFFFFF);

/* erase all  */
result = BDI_FlashErase(BDI_ERASE_CHIP, 0x00100000);

/* program/verify from a file */
result = BDI_FlashWriteFile("sam7s_64k.sss", &errorAddr);
result = BDI_VerifyFile("sam7s_64k.sss", &errorAddr);
```