ASSET
Driving Embedded Instrumentation

ScanWorks Embedded Diagnostics (SED)

API Library Reference Manual

Release 2.01

September 26, 2021

© 2021 ASSET InterTech, Inc.

Contents

Revision History 5

Introduction and Code Samples 6

ai_ErrorToString 15

ai_GetLibraryVersion 16

ai_mBasicRWRamTest 17

ai_mBootRomBusTest 20

ai_mCheckMemory 22

ai_mClose 25

ai_mConfig 26

ai_mCPUID 28

ai_mDownloadUserDiag 30

ai_mDRamRefreshTest 33

ai_mEnableoxmdebug 36

ai_mEnableRAMAreaasCAR 38

ai_mEnableUUTDiagsAreaasCAR 40

ai_mEnterDebugMode 42

ai_mExecuteUserDiag 44

ai_mExitDebugMode 47

ai_mFillMemory 49

ai_mFXRSTOR 52

ai_mFXSAVE 54

ai_mGetActiveCore 56

ai_mGetActiveCPU 57

ai_mGetActiveThread 58

ai_mGetBreakpoint 59

ai_mGetDebugModeStatus 61

ai_mGetITPScanChainTopology 63

ai_mGetPriorStateInfo 66

ai_mIOSFcrashdumpDiscovery 68

ai_mIOSFcrashdumpGetFrame 69

ai_mIOSFreadEndpointConfig 70

© 2021 ASSET InterTech, Inc.

ai_mIOSFreadMSR 71
ai_mIOSFreadPCIConfig 72
ai_mIOSFreadPCIConfigLocal 73
ai_mIOSFreadPkgConfig 74
ai_mIOSFTAPinit 75
ai_mIOSFTAPownership 76
ai_mIOSFwritePCIConfig 77
ai_mIOSFwritePCIConfigLocal 78
ai_mIOSFwritePkgConfig 79
ai_mIsPowerOn 80
ai_mNavigatetoTAPState 81
ai_mOpen 82
ai_mOpenEx 84
ai_mRamBusTest 86
ai_mRamBusTestChannel 89
ai_mRamBusTestviaFIFO 92
ai_mReadCR 95
ai_mReadCSR 97
ai_mReadDescriptorTableRegister 99
ai_mReadDR 101
ai_mReadGPR 103
ai_mReadIO 105
ai_mReadMemory 107
ai_mReadMSR 109
ai_mReadSegmentRegister 111
ai_mResetDetect 113
ai_mResetUUT 114
ai_mReturnIDCode 116
ai_mReturnIDCodewithOverscan 118
ai_mReturnSiliconID 120
ai_mRomCrcTest 122
ai_mRunUUT 125
ai_mRWRamTest 126

© 2021 ASSET InterTech, Inc.

ai_mScanDr 127
ai_mScanIr.. 130
ai_mSetActiveCore.. 132
ai_mSetActiveCPU.. 133
ai_mSetActiveThread.. 135
ai_mSetBreakpoint 136
ai_mSetDebugModeCheckFlag 138
ai_mSetinitbreak.. 139
ai_mSetmachinecheckbreak 141
ai_mSetRunMode 143
ai_mSetshutdownbreak.. 145
ai_mSetsmmentrybreak.. 147
ai_mSetTap 149
ai_mSetTargetCPUType 150
ai_mStopTest 152
ai_muregraw 153
ai_muregraw64 154
ai_mWaitforDebugMode 157
ai_mWBINVD 159
ai_mWriteCR 161
ai_mWriteCSR 163
ai_mWriteDescriptorTableRegister 165
ai_mWriteDR.. 167
ai_mWriteGPR.. 169
ai_mWriteIO.. 171
ai_mWriteMemory 173
ai_mWriteMSR 175
ai_mWriteSegmentRegister 177

© 2021 ASSET InterTech, Inc.

Revision History

Revision Number	Description	Date
2.01	- Removed NDA requirement, retained Copyright	September 26, 2021
2.00	- Sapphire Rapids support - CheckValue parameter added to ai_mCheckMemory() - ai_mGetBreakPoint() BreakPointNo parameter should be uint8_t* and not uint8_t - ai_mGetITPScanChainTopology: ai_CoreTopology_t core[24] should be core[60]	September 7, 2021
1.00	Original document	March 25, 2021

© 2021 ASSET InterTech, Inc.

Introduction and Code Samples

Welcome to our API Reference Manual! We hope that you find the contents useful and interesting. If at any time you have any feedback or questions, feel free to reach out to us at ai-info@asset-intertech.com, or use our handy Contact Us form at https://www.asset-intertech.com/contact-us/.

This API Reference Manual provides an alphabetical listing of all interfaces available within the ScanWorks Embedded Diagnostics (SED) run-control library. It is ASSET's intent to keep the library and documentation current, universal, and backwards compatible as of the most recent supported x86 platforms. As of the time of this writing, the current supported Intel CPU/platform is Sapphire Rapids/Eagle Stream, and the library and documentation support all the way back to the Intel Nehalem platform.

The two major use cases for the SED API are (1) remote CScripts application, and (2) On-Target Diagnostics (OTD). You can see this from the below diagram:

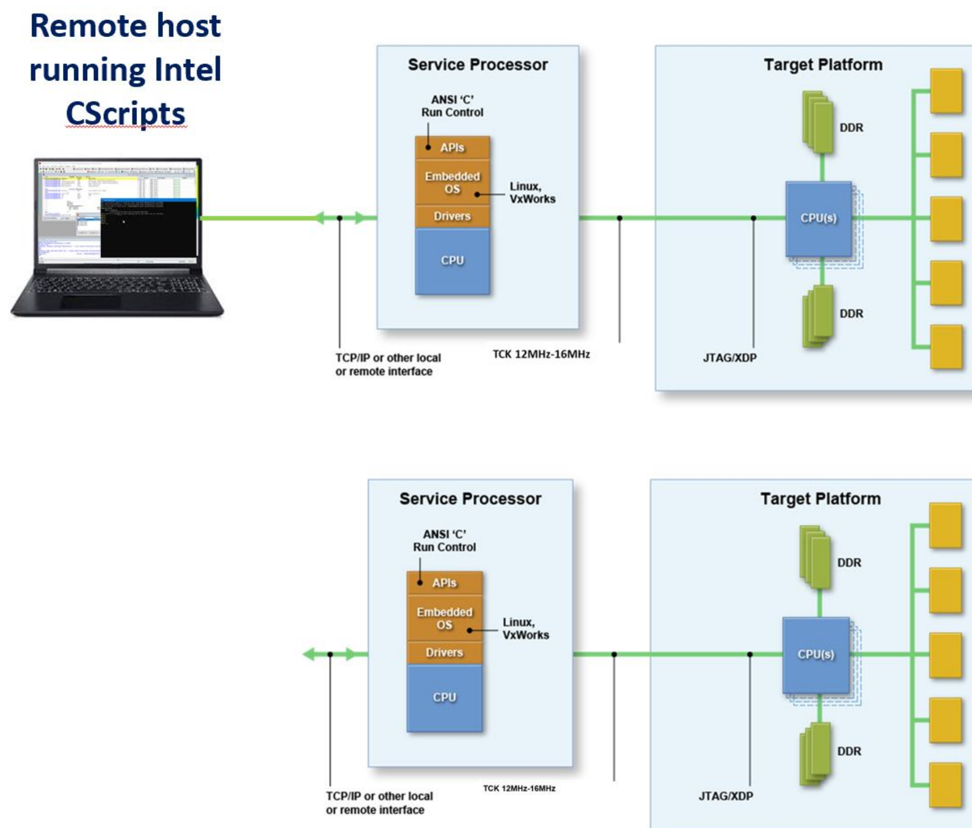

Figure 1: Remote CScripts, and On-Target Diagnostics

At the top, the remote host PC, running Windows or Linux, communicates over Ethernet or other transport with the BMC, that hosts the run-control library and the JTAG driver and mastering function.

© 2021 ASSET InterTech, Inc.

At the bottom is the main topic of this Reference Manual: the BMC is autonomous, and executes applications written in C/C++ as OTDs. We'll describe the general approach involved in using the SED library, and then walk through individual code samples to show how it's used.

Note: in this Reference Manual, although we strive to write efficient code and follow best SED programming practices, the main goal of each sample program is to demonstrate SED concepts or programming techniques. Writing the most optimal code was not the goal, and would likely obfuscate the ideas trying to be illustrated. Keep this in mind if you are using any of the sample code in your own projects, as you may wish to rework it for better efficiency. Moreover, in order to focus on the SED API, we have built minimal infrastructure on top of it. This means the illustrative software sometimes hardcodes values and define things in the source code that might normally be data driven.

We will illustrate the main functionality of the SED API by using a program called "`ltloop`". `ltloop` is an application that stresses PCI Express ports by exercising the Link Training & Status State Machine (LTSSM) on upstream and downstream PCI Express devices, looking for device, firmware and board marginalities. If you're unfamiliar with the LTSSM, please review some background reading such as [Built-In Self Test (BIST) for PCI Express using Embedded Run-Control] and [What is Surprise Link Down (SLD)?] We won't delve into the detailed nature of this application here, other than to use it for illustrative purposes for the API. Note that this example works on an Ice Lake server platform.

The source code for `ltloop` is here: [Download ltloopExample].

Let's firstly start off by looking at `main()`:

```
int main (int argc, char **argv)
{
    int numcores;
    int curcore;
    int numcpus;
    int curcpu;
    int iError = 0;
    bool pwrchk = true, scnsetup = true, savemodarch = true;
    int mHandle;
    FILE *UUTDiagsHexFile;
    char ver[200];
    uint64_t msr;
    uint64_t regdata;
    int i;
    struct timespec start_time;
    struct timespec end_time;
    double secs;
    uint32_t bus;
    uint32_t dev;
    uint32_t fun;
    ai_ITPtopology_t topo;
    uint16_t curCPU;

    UUTDiagsHexFile = NULL;

    printf("\n\nLink Training Loop test\n");

    iError = parseArgs(argc, argv);
    if (iError != 0)
    {
        usage();
        return iError;
    }

    ai_GetLibraryVersion(ver);
    printf("Library version = %s\n", ver);

    AI_pdcselector pdctarget = AI_pdc_0;

    if ((iError = ai_mOpen(pdctarget, 1, &mHandle)) != AI_SUCCESS)
    {
```

© 2021 ASSET InterTech, Inc.

```
        printf ("\nOpen ERROR: %s Channel %i\n" , ai_ErrorToString(iError), pdctarget);
        return 1;
    }

    if ((iError = ai_mSetTargetCPUType(mHandle, AI_sandybridge)) != AI_SUCCESS)
    {
        printf ("\nSetTargetCPUType: ERROR: %s Channel %i\n" , ai_ErrorToString(iError), pdctarget);
        return 1;
    }

    ai_mConfig (mHandle, 100, UUTDiagsHexFile, 0x10000LL, pwrchk, scnsetup, savemodarch);

    iError = ai_mGetITPScanChainTopology(mHandle, &topo, true);
    if (iError != AI_SUCCESS)
    {
      printf ("\nERROR getting target topology: %s\n" , ai_ErrorToString(iError));
      return iError;
    }

    numcpus = topo.tck[TCK_ZERO_POS].numcpus;
    if ((m_socket < CPU_ZERO_POS) || (m_socket > numcpus))
    {
        printf("Invalid socket number, must be between %hu and %hu\n", CPU_ZERO_POS, numcpus);
        return -1;
    }

    //Halt all cores in all CPUs
    for (curCPU=CPU_ZERO_POS; curCPU < (numcpus + CPU_ZERO_POS); curCPU++)
    {
        if ((iError = ai_mSetActiveCPU(mHandle, curCPU)) != AI_SUCCESS)
        {
            printf ("\nSetActiveCPU: ERROR: %s Socket %hu\n" , ai_ErrorToString(iError), curCPU);
            return 1;
        }

        ai_mSetActiveCore(mHandle, CORE_ZERO_POS);
        ai_mSetActiveThread(mHandle, THREAD_ZERO_POS);
        if ((iError = ai_mEnterDebugMode(mHandle)) != AI_SUCCESS)
        {
            printf ("\n EnterDebugMode: ERROR: %s Socket %hu\n" , ai_ErrorToString(iError), curCPU);
            return 1;
        }
    }

    //TODO
    //Check return values:
    ai_mIOSFTAPinit(mHandle);

    m_bus0 = 0;     //Start with bus 0 for bus discovery

    //Get TAP ownership for all TAPs, overview will use all TAPs (CPUs)
    for (curCPU=0; curCPU < numcpus; curCPU++)
    {
        m_peciCPU = curCPU;
        ai_mIOSFTAPownership(mHandle, true, curCPU);
    }

    //Prepare the target and get the bus numbers for each socket
    //We need the bus min/max for first socket in order for second socket to work
    for (curCPU=0; curCPU < numcpus; curCPU++)
    {
        m_peciCPU = curCPU;
        prepTarget(mHandle, 0, 0, 0);  //attempt to "unhide" devices; b/d/f is ignored
        getBusNumbers(mHandle, curCPU);
    }

    printf("Selecting socket %hu\n", m_socket);

    if ((iError = ai_mSetActiveCPU(mHandle, m_socket)) != AI_SUCCESS)
    {
        printf ("\nSetActiveCPU: ERROR: %s Socket %hu\n" , ai_ErrorToString(iError), m_socket);
        return 1;
    }
    m_peciCPU = m_socket - CPU_ZERO_POS;
    ai_mSetActiveCore(mHandle, CORE_ZERO_POS);
    ai_mSetActiveThread(mHandle, THREAD_ZERO_POS);


    clock_gettime(CLOCK_MONOTONIC, &start_time);

    port2bdf(m_port, &bus, &dev, &fun);     //Convert the command line option -p<n> to bus, device, function
```

© 2021 ASSET InterTech, Inc.

```
    do_test(mHandle, numcpus, bus, dev, fun);

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    secs = (double)(end_time.tv_sec - start_time.tv_sec) + (double)(end_time.tv_nsec - start_time.tv_nsec) / 10
00000000.0;
    printf("Time for test: %7.2f seconds.\n\n", secs);

    ai_mClose(mHandle);

    return iError;
}
```

Most OTDs have the same structure within the main routine. The platform is initialized for run-control with the following API, in sequential order:

```
ai_mOpen
ai_mSetTargetCPUType
ai_mConfig
ai_mGetITPScanChainTopology
ai_mSetActiveCPU
ai_mSetActiveCore
ai_mSetActiveThread
```

The operation of these functions should be made fairly clear by looking at the source code. Here are some examples.

The invocation of `ai_mOpen` is in line 1255:

```
if ((iError = ai_mOpen(pdctarget, 1, &mHandle)) != AI_SUCCESS)
```

Since the third argument is `&mHandle`, the value of `mHandle` can be updated within this function. `mHandle` is a unique identifier connection to a specific PdcNo – which in turns relates to the number of CPUs that the service processor JTAG Master is intended to drive. In this case, it's left at the default of `AI_PDC_0`; the default of one chain or node.

The invocation of `ai_mSetActiveCPU` is in line 1286:

```
if ((iError = ai_mSetActiveCPU(mHandle, curCPU)) != AI_SUCCESS)
```

while iterating across the values of `curCPU` (normally two in a 2-socket system, four in a 4-socket system, etc.).

And `ai_mEnterDebugMode` is in line 1294:

```
if ((iError = ai_mEnterDebugMode(mHandle)) != AI_SUCCESS)
```

This function forces all connected CPU cores of the node identified by `mHandle` in to debug mode. Said another way, they enter probe mode, and are then available for other run-control operations. Debug mode or probe mode is a state in which the platform must be to execute many of the SED library

© 2021 ASSET InterTech, Inc.

functions; though this is not true in all cases, which will be shown when we discuss the Intel On-chip System Fabric (IOSF) related functions, such as invoked in `ai_mIOSFTAPinit(mHandle)` on line 1303. But, let's skip ahead of these calls for now.

After the platform probe mode initialization is complete, the flow of the program is as follows:

```
prepTarget(mHandle, 0, 0, 0);
getBusNumbers(mHandle, curCPU);
port2bdf(m_port, &bus, &dev, &fun);
do_test(mHandle, numcpus, bus, dev, fun);
lt_loop(mHandle, bus, dev, fun);
```

After the main platform initialization is done, `ltloop` does some specific work to provide sideband (i.e. not run-control based) access with the IOSF routines in `main()`. Although there are references to PECI in the code, in this instance JTAG is still used as the physical access mechanism to the meta state machine, but not run-control. Then `port2bdf` is called (a handy utility function to translate the port number from the command line to bus/ device/ function). Finally `do_test` is launched, which in turn launches the `lt_loop` function, where all the heavy lifting is done.

Let's first look at the parseArgs(argc, argv) function called initially within main() to see the overall parameters and operations supported by the ltloop OTD:

```
int parseArgs(int argc, char **argv)
{
    int c;
    int retval = 0;
    while ((c = getopt (argc, argv, "l:p:t:s:f:cdbo?h")) != -1)
    {
        switch(c)
        {
            case 'p':
                //printf("Option p: %s\n", optarg);
                m_port = atoi(optarg);
                if ((m_port < 0) || (m_port >= MAXPORT))
                {
                    printf("Invalid port value: %d \n", m_port);
                    retval = -1;
                    m_port = 0;
                }
                break;

            case 'l':
                m_loops = atoi(optarg);
                if (m_loops < 1)
                {
                    printf("Invalid number of loops, must be > 0.\n");
                    retval = -1;
                    m_loops = 1;
                }
                break;

            case 's':
                m_socket = atoi(optarg);
                //Will check socket # in main after we get system topology and learn # of CPUs
                break;

            case 't':
                m_type = atoi(optarg);
                if ((m_type != 1) && (m_type != 6))
```

© 2021 ASSET InterTech, Inc.

```
                {
                    printf("Invalid test type, only type 1 and type 6 are supported.\n");
                    m_type = 1;
                    retval = -1;
                }
                break;

            case 'd':
                m_readDSC = true;
                break;

            case 'b':
                ai_mIOSFdebugPrint(true);
                printf("Debug printouts turned on\n");
                break;

            case 'o':
                m_overview = true;
                break;

            case 'f':
                m_forceSpeed = atoi(optarg);
                if ((m_forceSpeed > 4) || (m_forceSpeed < 1))
                {
                    printf("Invalid speed, must be between 1 and 4\n");
                    retval = 1;
                    m_forceSpeed = 0;
                }
                break;
            case 'c':
                m_pciScan = true;
                break;
            case '?':
            case 'h':
                retval = 1;     //Note: caller will treat as error and print usage()
                break;

        }  // switch...
    }  //while...

    //Further checks on params
    //Can't check downstream component on DMI (port 0)
    if ((m_port == 0) && (m_readDSC))
    {
        m_readDSC = false;
        printf("Unable to check DSC errors on DMI.\n");
        //We allow test to continue in this case, as it is probably what user would do if we issu
ed an error.
    }

    return retval;

}  // parseArgs
```

You can see that the switch on the input parameters (i.e. we might invoke ltloop via “ltloop -s1 -p5 -t1 -l10000”) gives us the following options:

“s” : socket number

“p”: PCIe port number

“l” : the number of LTSSM loop tests to run; for example, how many retrains to do

“t” : Test type. Only 1 and 6 are supported in this routine. “1” is a simple link retrain test, and “6” is a speed change loop test (i.e. from Gen1 to Gen4 and back).

© 2021 ASSET InterTech, Inc.

“d” : this provides the option to conduct testing on the downstream device (DSC – downstream component) as well as the upstream component. The default is “false” to optimize for time. But we can set it to “true” to have the test be more rigorous.

“b” : This allows for debug information to be sent from when we use the IOSF to access the PCI Express configuration registers. It’s a debug utility, mostly for use by ASSET or Intel personnel to do some troubleshooting if needed.

“o” : Another debug utility, just to do a subset equivalent of the “sysTopo” CScript. This overrides all other options.

“f” : Forces the port to a given speed; for example, Gen1, Gen2, Gen3, Gen4.

“c” : Does a scan and a print of all B/D/F.

Much of the “heavy lifting” happens in the `lt_loop(int mHandle, uint32_t bus, uint32_t device, uint32_t function)` routine. Note that it makes use of some of the key register definitions as defined early in the program:

```
#define REG_LNKSTS 0x52
#define REG_DMI_LNKSTS 0x1B2
#define REG_LNKCON 0x50
#define REG_DMI_LNKCON 0x1B0
#define REG_LNKCON2 0x70
#define REG_DMI_LNKCON2 0x1C0
#define REG_LNKCAP 0x4C
#define REG_LTRCON 0x11C
#define REG_CORERRSTS 0x110       //"ERRCORSTS" correctable error status
#define REG_UNCERRSTS 0x104        //"ERRUNCSTS" uncorrectable error status
#define REG_SECBUS 0x19
#define REG_AERCAPHDR 0x100       //Advanced Error Reporting Extended Capability Header
#define REG_CPUBUSNO 0x104
#define REG_CPUBUSNO1 0x108
#define REG_CPUBUSNO2 0x10A
#define REG_CPUBUSNO_VALID 0x110
#define REG_SOCKET_BUS_RANGE 0x114
```

The PCIe register definitions such as `REG_LNKSTS` are found in the Intel EDS; but, in lieu of that, a lot of the underlying methodology is common to all x86 platforms and in the public domain, such as here: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-processor-s1200-datasheet-vol-2.pdf. Note that `lnksts` has been renamed to `linksts`, but we keep the old legacy `LNKSTS` here. `lnksts` is at offset 0x52 and contains useful information such as whether link training is complete or in progress, what the link width is, whether it’s up or not, and the link speed. We see later that we read its contents with the call at line 621:

© 2021 ASSET InterTech, Inc.

```
lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg_lnksts, AI_bw16);
```

and then we do some compares, status checking, and place the link width and speed in our local variables:

```
if (lnksts == 0xFFFF)
    {
        printf("Link status is all ones; aborting test.\n");
        return;
    }
//check to make sure the link is active before we begin.  May be inactive due to not connected on target.
if ((lnksts & 0x2000) == 0)
    {
        printf("Link is not active on startup, aborting test.\n");
        return;
    }
startWidth = (lnksts >> 4) & 0x3F;
startSpeed = lnksts & 0xF;
```

Note at the top that we retrieve lnksts using the function `readRegisterPeci` as opposed to a standard JTAG/MMIO read. This routine does not use PECI as an access mechanism; rather, it uses the sideband IOSF access mechanism as a means to not halt the target. So, despite the name, it's still JTAG. The end result is the same, though: we retrieve all 16 bits of `LNKSTS`, do the "and" with bit 13 to see if the Data Link Control and Management State Machine is in the `DL_Active` state, and if so we get the Negotiated Link Width by shifting right four bits and ANDing with x3F, thereby isolating the bits 9:4 which contains the width. We also assign startSpeed to the last four bits.

Next we do the same thing with `lnkcon` (in the EDS, this has been renamed to `linkctl`):

```
lnkcon = readRegisterPeci(mHandle, bus, dev, fun, reg_lnkcon, AI_bw16);\
lnkcon = lnkcon | 0x20; //set the retrain_link bit
```

Note that we set bit 5 of `lnkcon`, via the OR with x20, to initiate the link retrain. We then later use run-control (not IOSF) to fire off the retrain:

```
//Write the LNKCON register to cause the link to retrain
writeRegisterMem(mHandle, membus, dev, fun, reg_lnkcon, AI_bw16, lnkcon);
```

We then iteratively loop for up to 100 times (normally it only takes one or two loops), checking the link retrain bit, to see if the retrain succeeded:

```
//loop checking the link status until the link has finished retraining (or we give up)
for (j=0; j<100; j++)   //max number of times to try
        {
            lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg_lnksts, AI_bw16);
            if ((lnksts & 0x800) == 0)
            {
                break;  //Normal exit for this loop after 1 or 2 tries
            }
        }
```

© 2021 ASSET InterTech, Inc.

```
if ((lnksts & 0x800) != 0)
        {
            printf("Link failed to finish re-training.\n");
            noTrains++;
            break;
        }
if ((lnksts & 0x2000) == 0)
        {
            printf("Link is no longer active after retrain.\n");
            noTrains++;
            break;
        }
```

And afterwards we check that the Link Training bit (bit 11 of `lnksts`, the result of the mask with 0x800) should be zero, and that we're in the `DL_Active` state (via the mask with 0x2000, bit 13). Otherwise, we bail.

Finally, as an example, we collect the number of correctable and uncorrectable errors while we're in the loop:

```
//Check for uncorrectable errors that occurred during retrain
errs = readRegisterPeci(mHandle, bus, dev, fun, REG_UNCERRSTS, AI_bw32);
if (errs != 0)
    {
        uncorErrors++;
        printf("Uncorrectable error: 0x%08X on port: %d loop: %d\n", errs, m_port, i);
    }
//Check for correctable errors that occured during retrain
errs = readRegisterPeci(mHandle, bus, dev, fun, REG_CORERRSTS, AI_bw32);
if (errs != 0)
    {
        corErrors++;
         printf("Correctable error: 0x%08X on port: %d loop: %d\n", errs, m_port, i);
    }
```

And that's it! Hopefully you've found this helpful in creating your own OTDs.

© 2021 ASSET InterTech, Inc.

NAME

`ai_ErrorToString`

Converts an integer error code to a string.

SYNOPSIS

```
#include <itp_driver.h>

const char *ai_ErrorToString(int rval);
```

DESCRIPTION

`ai_ErrorToString()` takes the integer error code `rval` and returns the associated character array of error information.

RETURN VALUE

The `ai_ErrorToString()` function returns a pointer to related error string.

ERRORS

N/A

© 2021 ASSET InterTech, Inc.

NAME

ai_GetLibraryVersion

Returns the version number of the shared library.

SYNOPSIS

```
#include <itp_driver.h>

int ai_GetLibraryVersion (char* version);
```

DESCRIPTION

`ai_GetLibraryVersion()` returns the version number of the ITP driver library through the version pointer to char.

RETURN VALUE

On success, `ai_GetLibraryVersion()` returns 0.

ERRORS

N/A

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

ai_mBasicRWRamTest

Execute a Basic R/W RAM Test diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mBasicRWRamTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, char* ErrorString);
```

DESCRIPTION

`ai_mBasicRWRamTest()` executes a test to diagnose the data and address buses between the CPU and a RAM area.

`mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

`ai_mBasicRWRamTest()` assumes a 64-bit data bus width (for data bus, byte enable lane and data cell testing).

If an error is diagnosed, `ErrorString` will return the diagnostic information.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

`ai_mBasicRWRamTest()` is divided in to 5 sub-tests, executed in sequence.

© 2021 ASSET InterTech, Inc.

1. Data bus hi/lo test.
2. Data bus shorts test.
3. Byte enables test.
4. Data test (writes/verifies patterns to every individual cell in the range specified).
5. Address bus test.

If any sub-test fails the diagnostic returns immediately, skipping execution of any subsequent sub-tests.

The 'Data test' sub-test uses machine code routines that operate in 32-bit mode only. Therefore, this diagnostic can only operate on the bottom 4G memory space (i.e. 0x0-0xFFFFFFFF).

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mBasicRWRamTest()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCOD was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: Error configuring JTAG
`AI_SA_GREATER_THAN_EA`	Start address cannot be greater than end address
`AI_BASIC_RW_RAM_DATA_HILO`	Data bus hi/lo failure diagnosed
`AI_BASIC_RW_RAM_DATA_SHORT`	Data bus shorts failure diagnosed
`AI_BASIC_RW_RAM_BYTE_ENABLES`	Byte enables lane(s) failure diagnosed
`AI_BASIC_RW_RAM_DATA_TEST`	Data cell test failure diagnosed
`AI_BASIC_RW_RAM_ADDRESS`	Address bus failure diagnosed
`AI_RAM_TEST_HALTED_USR`	Execution of RAM test interrupted by user
`AI_RAM_TEST_HALTED_UNKNWN_SRC`	Execution of RAM test interrupted by unknown source

© 2021 ASSET InterTech, Inc.

`AI_ERR_PCT_INT_IOCTL`	Error reported from `IOCTL()` function
`AI_FILE_LOAD_ERR`	Error loading hex file (required for data cell test)
`AI_DBG_MODE_RAM_TEST`	Error during execution of RAM Test
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG/Configuration settings to default

SEE ALSO

ai_mRamBusTest

ai_mRamBusTestChannel

ai_mRamBusTestviaFIFO

ai_mRWRamTest

ai_mDRamRefreshTest

© 2021 ASSET InterTech, Inc.

NAME

`ai_mBootRomBusTest`
Execute a Boot ROM Bus Test diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mBootRomBusTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, char* ErrorString);
```

DESCRIPTION

`ai_mBootRomBusTest()` executes a test to diagnose the data and address buses between the CPU and the boot ROM area. Because of the nature of ROM memory (i.e. read only), only read memory operations can be used to diagnose the buses.

`mHandle` identifies the node for the operation to be carried out on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mBasicRWRamTest()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

`ai_mBootRomBusTest()` is divided in to 2 sub-tests, executed in sequence.

1. Data bus test.

2. Address Bus Test.

© 2021 ASSET InterTech, Inc.

If any sub-test fails, the diagnostic returns immediately, skipping execution of any subsequent sub-tests.

The test does not necessarily have to be execute on a ROM area of memory. It can be executed on a RAM area. However, the target range must have random data within it (the diagnostic algorithms assume random data in the range).

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mBootRomBusTest()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_SA_GREATER_THAN_EA`	Start address cannot be greater than end address
`AI_ROM_BUS_FAIL`	Bus test diagnosed a failure
`AI_ROM_TEST_HALTED_USR`	Execution of Boot ROM bus test interrupted by user
`AI_DBG_MODE_ROM_BUS`	Error during execution of Boot ROM bus test
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG/configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mCheckMemory`

Check that a memory block/range contains a specific value

SYNOPSIS

```
#include <itp_driver.h>
```

```
int ai_mCheckMemory (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, void* FillValue, void* CheckValue, ai_Buswidth BusWidth,
uint64_t* ErrorAddress, void* ErrorData);
```

DESCRIPTION

`ai_mCheckMemory()` submits instruction(s) to the target core on the node specified by `mHandle`, to execute the 'check memory' machine code routine on the target core, to check a memory block starting at `StartAddress` and ending at `EndAddress` for the value `CheckValue`. As each location is checked, it will be overwritten with the value `FillValue`.

`BusWidth` takes on one of the following values to specify the operand size for the operation.

8 8-bit operand size

16 16-bit operand size.

32 32-bit operand size

`ErrorAddress` will return the memory location in error on a failure occurrence, and `ErrorData` will return the data at the failing memory location.

During execution of the 'check memory' machine code routine, the target core will exit debug mode. The function will not return until the core has re-entered debug mode. Upon completion, the target core should immediately re-enter debug mode, at which point, the target core can then service ITP driver functions normally again.

Should the user wish to force re-entry to debug mode during execution of the 'check memory' machine code routine, he/she can do so by calling `ai_mStopTest()` via a forked child process. `ai_mStopTest()` will force debug mode re-entry, which will cause `ai_CheckMemory()` to subsequently return.

No other ITP Driver functions (other than `ai_mStopTest()`) should be called while `ai_mCheckMemory()` is running. Because the target core is not in debug mode during user diagnostic execution, execution of other ITP Driver functions can force the target core to re-enter debug mode. In this case, `ai_mCheckMemory()` behavior is undefined.

© 2021 ASSET InterTech, Inc.

The 'check memory' machine code routine forms part of a collection of machine code routines which the ITP driver can execute. Since the routines are machine code, they must be downloaded and run from an area of memory accessible by the target core. `UUTDiagsHexFile from ai_mConfig()` provides the ITP driver library with a pointer to the machine code file, and `UUTDiagsBaseAddress`, also from `ai_mConfig()`, defines the memory base address from which the machine code will be run. Prior to calling the 'check memory' machine code routine, the function will check if the machine code routines exists at `UUTDiagsBaseAddress`, and if not, will proceed to download the file pointed to by `UUTDiagsHexFile`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The machine code routines operate in 32-bit mode only, therefore any machine code routines will only operate in the bottom 4G memory space (i.e 0x0-0xFFFFFFFF). Behavior is undefined if the range specified extends outside this area.

When calling `ai_mCheckMemory()`, the user should ensure that memory range specified does not overlap into the `UUTDiagsBaseAddress` memory area reserved for execution of the machine code routines. Also, the user should ensure that memory has been initialized sufficiently to allow the machine code routines to run properly. In both cases, the function may fail to return normally (i.e. unless forced using `ai_StopTest()`).

RETURN VALUE

On success, `ai_mCheckMemory()` returns 0. On error, it will return one of the following values:

ERRORS

© 2021 ASSET InterTech, Inc.

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: rrror getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_CHECK_MEM	Error during execution of check memory operation
AI_CHECK_MEM_FAIL	Memory failure
AI_CHECK_MEM_HALTED_USR	Execution of check memory operation interrupted by user
AI_CHECK_MEM_HALTED_UNKNWN_SRC	Execution of check operation interrupted by unknown source
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address
AI_NO_EXEC_HALT_STATE	Target is halted. Unable to execute user diagnostic
AI_NO_EXEC_WAIT_FOR_SIPI_STATE	Target indicates 'WAIT FOR SIPI' state. Unable to run diagnostic
AI_NO_EXEC_SHUTDOWN_STATE	Target in 'SHUTDOWN' state. Unable to run diagnostic
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

FILES

Pentcode.hex - machine code routines collection. Can be installed to any directory. UUTDiagsHexFile from ai_mConfig() provides the ITP driver library with a pointer to the machine code file.

SEE ALSO

ai_mFillMemory

ai_mStopTest

NAME

© 2021 ASSET InterTech, Inc.

ai_mClose

Close communication channel with ITP Driver JTAG Controller device.

SYNOPSIS

```
#include <itp_driver.h>

void ai_mClose (int mHandle);
```

DESCRIPTION

`ai_mClose()` closes the ITP Driver 'connection' specified by `mHandle`.

The function submits a reset to the ITP driver JTAG controller device, thereby relinquishing control over the XDP and JTAG lines. It then proceeds to unmap the ITP Driver JTAG controller from memory space, and finally closes the file handle to the device.

RETURN VALUE

N/A

ERRORS

N/A

SEE ALSO

`ai_mOpen`

`ai_mOpenEx`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mConfig`
Set up configuration parameters.

SYNOPSIS

```
#include <itp_driver.h>
```

```
int ai_mConfig (int mHandle, uint16_t ResetPulseDuration, FILE
*UUTDiagsHexFile, uint64_t UUTDiagsBaseAddress, bool PowerCheck, bool
ScanChainSetup, bool SaveModifyArch);
```

DESCRIPTION

`ai_mConfig()` sets up various configuration parameters for the node identified by `mHandle` that may be required by other ITP driver library function calls.

`ResetPulseDuration` specifies the length of pulse (in mSecs) to be applied to the DBR line (XDP interface - HOOK7) when the `ai_mResetUUT()` or `ai_mRunUUT()` functions are invoked.

`UUTDiagsHexFile` provides a file pointer to the UUT diagnostics hex file. The UUT diagnostics hex file contains the machine code language routines that are used by the ITP driver library to carry out ROM CRC checks, fill memory commands, and check memory commands. Some RAM test diagnostics also have fill and check memory commands built into their algorithms.

`UUTDiagsBaseAddress` provides the base address in memory where the UUT diagnostics hex file will be downloaded and executed from. Diagnostics can only be downloaded and/or executed from the bottom 4GB of memory (0x0 - 0xffffffff).

`PowerCheck` provides a global parameter to the other ITP driver library functions informing them to execute a power check first, before continuing to execute their designated function. Default value is `true` (i.e. do a power check)

`ScanChainSetup` provides a global parameter to the other ITP driver library functions informing them to check and perform (if necessary) the required scan chain interrogation and setup before executing their designated function. Default value is `true` (i.e. check and perform scan-chain interrogation/set-up).

`SaveModifyArch` provides a global parameter to the other ITP driver library functions informing them to check and perform (if necessary) the debug mode entry and processor state save routines before executing their designated function. Most of the ITP driver functions require the target CPU to be in debug mode, and the state of the processor at the debug mode entry point to be saved to a buffer within the ITP driver. Default value is `true` (i.e. check and perform debug mode entry and save processor state routines).

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mConfig()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mCPUID`
Submit a CPUID instruction and retrieve CPUID info from target core.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mCPUID (int mHandle, uint32_t inEAX, uint32_t inECX, uint32_t
*retEAX, uint32_t *retEBX, uint32_t *retECX, uint32_t *retEDX);
```

DESCRIPTION

`ai_mCPUID()` submits a CPUID instruction to the target core, on the node specified by `mHandle`, with the data presented in `inEAX` and `inECX` as the inputs. The output from the execution of the CPUID instruction is saved and returned in `retEAX` , `retEBX` , `retECX` and `retEDX` . See Intel 64 and IA-32 Architectures Software Developer Manual Volume 2A for more information on the CPUID function.

On some processors CPUID functionality is not available. Prior to submitting the CPUID instruction for execution, the function performs an operation to check if such functionality is available.

The input and output parameters do not have any effect on the 'processor state buffer' held in host memory. (i.e. the target core 'processor stat buffer' remains unaffected through the execution of this function).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mCPUID()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible UNCORES exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_CPUID_NOT_AVAIL`	Target core reports CPUID functionality not available
`AI_DBG_MODE_CPUID`	Error during execution of CPUID operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mDownloadUserDiag`

Download a user diagnostic to target memory.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mDownloadUserDiag (int mHandle, FILE *userdiagfilefd, uint64_t
BaseAddress);
```

DESCRIPTION

`ai_mDownloadUserDiag()` submits instruction(s) to the target core, on the node specified by `mHandle`, to download the data from the binary file pointed to by `userdiagfilefd`, to a memory area with the base address location specified by `BaseAddress`.

The file pointed to by `userdiagfilefd` must be of Intel HEX format. Normally, the code for such Intel HEX format is written in assembly language and compiled using a macro assembler of some description, such as the Microsoft Macro Assembler (MASM).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

On debug mode entry (and after saving the architectural state), the target core will be in placed in 32-bit operating mode. As shown in the example below, user diagnostics should assume 32-bit mode operation. Based upon the 32-bit operation assumption, it is not possible for user diagnostics to execute above the 4GB boundary (0xffffffff). Any attempt to download user diagnostics using BaseAddress greater than 0xffffffff will be rejected. Users should ensure their user diagnostic(s) do not extend

© 2021 ASSET InterTech, Inc.

beyond the 4GB boundary. In this case, no error is produced, but execution of the user diagnostic will produce undefined behavior.

RETURN VALUE

On success, `ai_mDownloadUserDiag()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_DWNLD_USR_DIAG`	Error during execution of user diagnostic
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

EXAMPLE

The code below demonstrates the format of a user diagnostic, and how it is written using assembly language such that it can be properly invoked by the `ai_mExecuteUserDiag()` call, and return the target core in a halted state (debug mode) on completion of the routine.

```
.386P
CODE  SEGMENT USE32

ASSUME CS:CODE,DS:CODE
BreakPoint: JMP   Breakpoint  ;* First line (obligatory!)
                  MOV   ECX,0000B8000H    ;* code
                  MOV   EAX,0ABCDEF10H    ;* code MOV [ECX],EAX ;* code
                  JMP   BreakPoint        ;* Last line (obligatory!)
;*******************************************************
;CODE ENDS HERE.
```

© 2021 ASSET InterTech, Inc.

```
;****************************************************** CODE ENDS

END
```

Note that the first and last lines are marked as obligatory. These lines must be present in the diagnostic otherwise it may not run or return to debug mode after completion. The user should insert the diagnostic routine between these two lines.

Although this example illustrates moving values into GPRs, data can be passed into and returned from the diagnostic by calling the `ai_mReadGPR()` and `ai_mWriteGPR()` prior to or after execution of the diagnostic.

SEE ALSO

`ai_mExecuteUserDiag`

© 2021 ASSET InterTech, Inc.

NAME

ai_mDRamRefreshTest

Execute a DRAM Refresh Test diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mDRamRefreshTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, double RefreshDelay, char* ErrorString);
```

DESCRIPTION

`ai_mDRamRefreshTest()` executes a test to find and diagnose refresh problems on a DRAM area.

`mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

`RefreshDelay` specifies the delay (in secs) between filling and checking the memory range, in each pass of the diagnostic.

If an error is diagnosed, `ErrorString` will return the diagnostic information.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The `ai_mDRamRefreshTest()` function uses machine code routines that operate in 32-bit mode only. Therefore, this diagnostic can only operate on the bottom 4G memory space (i.e. 0x0-0xFFFFFFFF).

© 2021 ASSET InterTech, Inc.

This is a 2-pass test. The first pass writes/verifies zeroes (0's) to each cell location, while the 2nd pass writes/verifies ones (1's) to each cell location.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mDRamRefreshTest()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address
AI_DRAM_REFRESH	DRAM refresh failure diagnosed
AI_RAM_TEST_HALTED_USR	Execution of RAM Test interrupted by user
AI_RAM_TEST_HALTED_UNKNWN_SRC	RAM test interrupted by unknown source
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_FILE_LOAD_ERR	Error loading hex file (required for test)
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_DBG_MODE_RAM_TEST	Error during execution of RAM test
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mRamBusTest

ai_mRamBusTestChannel

ai_mRamBusTestviaFIFO

ai_mBasicRWRamTest

© 2021 ASSET InterTech, Inc.

ai_mDRamRefreshTest

`ai_mRWRamTest`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mEnableoxmdebug`
Enable target ureg_raw capability

SYNOPSIS

```
#include <itp_ureg_raw.h>

int ai_mEnableoxmdebug (int mHandle);
```

DESCRIPTION

`ai_mEnableoxmdebug()` enables the target CPU, for the node identified by `mHandle`, to be receptive to `uregraw` type scans. The target is briefly put in to debug mode to carry out the operation, after which debug mode is exited, and the target is left in a running state again. This function is provided mainly as an interface for Intel CScripts. THIS API IS ONLY APPLICABLE TO INTEL IVY BRIDGE TARGETS.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success, `ai_mEnableoxmdebug()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP

© 2021 ASSET InterTech, Inc.

AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_UREG_RAW_NOT_IMPLEMENTED	Not a valid function (possibly not IvyBridge target)
AI_DBG_MODE_ENABLE_OXM_DEBUG	Error during execution of Enableoxmdebug operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_muregraw

ai_muregraw64

© 2021 ASSET InterTech, Inc.

NAME

`ai_mEnableRAMAreaasCAR`
Set up a RAM area as cacheable.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mEnableRAMAreaasCAR (int mHandle, uint64_t CarBaseAddress,
uint64_t CarSize);
```

DESCRIPTION

`ai_mEnableRAMAreaasCAR()` submits instruction(s) to the target core, on the node specified by `mHandle`, to invalidate its cache, and set up an area of target memory as cacheable. CarSize defines the size of the area to be defined as cacheable, and CarBaseAddress defines the base address of the area to be made cacheable.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

CAR can only be defined within the bottom 4GB memory region (i.e. 0x0 - 0xffffffff).

RETURN VALUE

On success, `ai_mEnableRAMAreaasCAR()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_SETUP_CAR_RAM`	Error during setup of RAM area
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mEnableUUTDiagsAreaasCAR`
Set up area from where machine code routines are run from as cacheable.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mEnableUUTDiagsAreaasCAR (int mHandle);
```

DESCRIPTION

`ai_mEnableUUTDiagsAreaasCAR()` submits instruction(s) to the target core, on the node specified by `mHandle`, to invalidate its cache, and set up a 4KB area around the machine code routines as cacheable. All memory accesses (other than to the cacheable machine code area) will then be directed to physical memory, as opposed to cache.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mEnableUUTDiagsAreaasCAR()` returns 0. On error, it will return one of the following values

ERRORS

© 2021 ASSET InterTech, Inc.

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_SETUP_CAR_PENTCODE	Error during setup of UUT diags area for CAR
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mEnableRAMAreaasCAR

© 2021 ASSET InterTech, Inc.

NAME

`ai_mEnterDebugMode`

Force all connected CPU core(s) into debug mode.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mEnterDebugMode (int mHandle);
```

DESCRIPTION

`ai_mEnterDebugMode()` forces all connected CPU cores on the node identified by `mHandle` into debug mode. This will halt all connected CPU core(s) at their current state. Debug mode is a temporary state, which the CPU core must be in to execute most of the ITP Driver library functions.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

RETURN VALUE

On success completion of the sequence, `ai_mEnterDebugMode()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG

© 2021 ASSET InterTech, Inc.

AI_DBG_MODE_ENTER	Error during execution to put CPU cores into debug more
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mExitDebugMode

ai_mGetDebugModeStatus

© 2021 ASSET InterTech, Inc.

NAME

ai_mExecuteUserDiag

Execute a user diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mExecuteUserDiag (int mHandle, uint64_t BaseAddress);
```

DESCRIPTION

`ai_mExecuteUserDiag()` submits instruction(s) to the target core, on the node identified by `mHandle`, to execute the user diagnostic with the base address location specified by `BaseAddress`.

During execution of the user diagnostic, the target core will exit debug mode. The function will not return until the core has re-entered debug mode. Upon completion of the diagnostic, the target core should immediately re-enter debug mode, at which point, the target core can then service ITP driver functions normally again.

Should the user wish to force re-entry to debug mode during execution of the user diagnostic, he/she can do so by calling `ai_mStopTest()` via a forked child process. `ai_mStopTest()` will force debug mode re-entry, which will cause `ai_mExecuteUserDiag()` to subsequently return.

No other ITP Driver functions (other than `ai_mStopTest()`) should be called while `ai_mExecuteUserDiag()` is running. Because the target core is not in debug mode during user diagnostic execution, execution of other ITP Driver functions could force the target core to re-enter debug mode. In this case, `ai_mExecuteUserDiag()` behavior is undefined.

Data can be passed into and returned from the diagnostic by calling the `ai_mReadGPR()` and `ai_mWriteGPR()` prior to, or after execution, of the diagnostic.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously)

© 2021 ASSET InterTech, Inc.

to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

On debug mode entry (and after saving the architectural state), the target core will be in placed in 32-bit operating mode. User diagnostics should assume 32-bit mode operation. Based upon the 32-bit operation assumption, it is not possible for user diagnostics to execute above the 4GB boundary (0xffffffff). Any attempt to execute user diagnostics using `BaseAddress` greater than 0xffffffff will be rejected. Users should ensure their user diagnostic(s) do not extend beyond the 4GB boundary. In this case, no error is produced, but execution of the user diagnostic will produce undefined behavior.

RETURN VALUE

On success, `ai_mExecuteUserDiag()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_EXEC_USR_DIAG`	Error during execution of user diagnostic
`AI_EXEC_USR_DIAG_HALTED_USR`	Error of user diagnostic operation interrupted by user
`AI_NO_EXEC_HALT_STATE`	Target core is in HALT state – cannot execute user diagnostic
`AI_NO_EXEC_WAIT_FOR_SIPI_STATE`	Target core is in WAIT FOR SIPI state – cannot execute user diagnostic
`AI_NO_EXEC_SHUTDOWN_STATE`	Target core is in SHUTDOWN state – cannot execute user diagnostic
`AI_ERR_PCT_INT_IOCTL`	Error reported from `IOCTL()` function
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

© 2021 ASSET InterTech, Inc.

ai_mExecuteUserDiag

`ai_mDownloadUserDiag`

`ai_mStopTest`

© 2021 ASSET InterTech, Inc.

NAME

ai_mExitDebugMode

Force all connected CPU core(s) out of debug mode.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mExitDebugMode (int mHandle);
```

DESCRIPTION

`ai_mExitDebugMode()` forces all connected CPU cores on the node identified by `mHandle` out of debug mode, if they are in debug mode. This will restart all connected CPU core(s) from their current debug mode state. The function also primes the ITP Driver FPGA controller device PRDY interrupt mechanism, to allow any subsequent PRDY pulses (indicative of debug mode activity, such as breakpoints) to be detected.

Before attempting the exit debug mode operation, each individual core's “processor state buffer” will be restored (if it has been saved/extracted from the core).

Also, before attempting the exit debug mode operation, the ITP Driver controller device is primed to halt the CPU core(s) on detection of any subsequent activity on the CPU RESET signal (i.e. a ' debug mode at reset condition'). However, execution of `ai_mClose()` will negate the ability to halt the CPU core(s) on the occurrence of a CPU RESET.

Debug mode status is not checked on completion of the exit debug mode operation. (i.e. the function does not check that all connected CPU core(s) have exited debug mode).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

RETURN VALUE

On success completion of the sequence, `ai_mExitDebugMode()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_STS`	Error during execution of get debug mode status
`AI_DBG_MODE_PROC_REGS`	Error during execution of restoring CPU core registers
`AI_DBG_MODE_WAKEUP_PREQ`	Error during execution of WAKEUP_PREQ operation
`AI_ERR_PCT_INT_IOCTL`	Error from `IOCTL()` function
`AI_DBG_MODE_EXIT`	Error during exit debug mode operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mEnterDebugMode`

`ai_mGetDebugModeStatus`

`ai_mWaitForDebugMode`

© 2021 ASSET InterTech, Inc.

NAME

ai_mFillMemory

Fill a memory block/range with a specific value.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mFillMemory (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, void* FillValue, AI_buswidth BusWidth);
```

DESCRIPTION

`ai_mFillMemory()` submits instruction(s) to the target core on the node specified by `mHandle`, to execute the ’fill memory' machine code routine on the target core, to fill a memory block starting at `StartAddress` and ending at `EndAddress` with the value FillValue.

`BusWidth` takes on one of the following values to specify the operand size for the operation:

8 8-bit operand size

16 16-bit operand size.

32 32-bit operand size.

During execution of the ’fill memory' machine code routine, the target core will exit debug mode. The function will not return until the core has re-entered debug mode. Upon completion, the target core should immediately re-enter debug mode, at which point, the target core can then service ITP driver functions normally again.

Should the user wish to force re-entry to debug mode during execution of the ’fill memory' machine code routine, he/she can do so by calling `ai_mStopTest()` via a forked child process. `ai_mStopTest()` will force debug mode re-entry, which will cause `ai_mFillMemory()` to subsequently return.

No other ITP Driver functions (other than `ai_mStopTest()`) should be called while `ai_mFillMemory()` is running. Because the target core is not in debug mode during user diagnostic execution, execution of other ITP Driver functions can force the target core to re-enter debug mode. In this case, `ai_mFillMemory()` behavior is undefined.

The ’fill memory' machine code routine forms part of a collection of machine code routines which the ITP driver can execute. Since the routines are machine code, they must be downloaded and run from an area of memory accessible by the target core. `UUTDiagsHexFile` from `ai_mConfig()` provides the ITP driver library with a pointer to the machine code file, and `UUTDiagsBaseAddress`, also from `ai_mConfig()`, defines the memory base address from which the machine code will be run. Prior to calling the ’fill memory' machine code routine, the function will check if the machine code routines exist

© 2021 ASSET InterTech, Inc.

at UUTDiagsBaseAddress, and if not, will proceed to download the file pointed to by UUTDiagsHexFile.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the PowerCheck option is not disabled via ai_mConfig()), the function will first of all perform a power check on the target.

Also, by default (if the ScanChainSetup option is not disabled via ai_mConfig()), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the SaveModifyArch option is not disabled via ai_mConfig()), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The machine code routines operate in 32-bit mode only, therefore any machine code routines will only operate in the bottom 4G memory space (i.e 0x0-0xFFFFFFFF). Behavior is undefined if the range specified extends outside this area.

When calling ai_mFillMemory(), the user should ensure that memory range specified does not overlap into the UUTDiagsBaseAddress memory area reserved for execution of the machine code routines. Also, the user should ensure that memory has been initialized sufficiently to allow the machine code routines to run properly. In both cases, the function may fail to return normally (i.e. unless forced using ai_mStopTest()).

RETURN VALUE

On success, ai_mFillMemory() returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected

© 2021 ASSET InterTech, Inc.

AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_FILL_MEM	Error during execution of fill memory operation
AI_FILL_MEM_HALTED_USR	Execution of fill memory operation interrupted by user
AI_FILL_MEM_HALTED_UNKNWN_SRC	Execution of fill memory operation interrupted by unknown source
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address
AI_NO_EXEC_HALT_STATE	Target core in HALT state – unable to execute user diagnostic
AI_NO_EXEC_WAIT_FOR_SIPI_STATE	Target core in WAIT FOR SIPI state – unable to execute user diagnostic
AI_NO_EXEC_SHUTDOWN_STATE	Target core in SHUTDOWN state – unable to execute user diagnostic
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

FILES

Pentcode.hex - machine code routines collection. Can be installed to any directory. UUTDiagsHexFile from ai_mConfig() provides the ITP driver library with a pointer to the machine code file.

SEE ALSO

ai_mCheckMemory

ai_mStopTest

© 2021 ASSET InterTech, Inc.

NAME

ai_mFXRSTOR

Execute the Pentium FXRSTOR instruction.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mFXRSTOR (int mHandle, uint64_t BaseAddress);
```

DESCRIPTION

`ai_mFXRSTOR()` restores the x87 FPU, MMX technology, XMM and MXCSR registers of the currently targeted core, on the node identified by `mHandle`, from a 512-byte memory area, the base of which is specified by `BaseAddress`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mFXRSTOR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter

© 2021 ASSET InterTech, Inc.

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_CPU_FUNC_NOT_SUPPORTED	Function not supported on targeted CPU
AI_DBG_MODE_FXRSTOR	Error during execution of FXRSTOR operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mFXSAVE

© 2021 ASSET InterTech, Inc.

NAME

ai_mFXSAVE

Execute the Pentium FXSAVE instruction.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mFXSAVE (int mHandle, uint64_t BaseAddress);
```

DESCRIPTION

`ai_mFXSAVE()` saves the x87 FPU, MMX technology, XMM and MXCSR registers of the currently targeted core, on the node identified by `mHandle`, to a 512-byte memory area, the base of which is specified by `BaseAddress`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), the function will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re- execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mFXSAVE()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter

© 2021 ASSET InterTech, Inc.

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_CPU_FUNC_NOT_SUPPORTED	Function not supported on targeted CPU
AI_DBG_MODE_FXRSTOR	Error during execution of FXRSTOR operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mFXRSTOR

© 2021 ASSET InterTech, Inc.

NAME

`ai_mGetActiveCore`
Get the currently targeted core.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetActiveCore (int mHandle, uint16_t *ActiveCore);
```

DESCRIPTION

`ai_mGetActiveCore()` retrieves the core that is currently being targeted by the ITP driver, on the node identified by `mHandle`, and returns it in `ActiveCore`.

NOTES:

The `ActiveCore` will always return to the default value (1) after `ai_mResetUUT()` , `ai_mRunUUT()`, or, on first loading of the ITP driver library to memory.

RETURN VALUE

`ai_mGetActiveCore()` will always returns 0.

ERRORS

N/A

SEE ALSO

`ai_mGetActiveCPU`

`ai_mGetActiveThread`

`ai_mSetActiveCPU`

`ai_mSetActiveCore`

`ai_mSetActiveThread`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mGetActiveCPU`

Get the currently targeted CPU.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetActiveCPU (int mHandle, uint16_t *ActiveCPU);
```

DESCRIPTION

`ai_mGetActiveCPU()` retrieves the CPU that is currently being targeted by the ITP driver, on the node identified by `mHandle`, and returns it in `ActiveCPU`.

NOTES:

The active CPU will always return to the default value (1) after `ai_mResetUUT()`, `ai_mRunUUT()`, or, on first loading of the ITP driver library to memory.

RETURN VALUE

`ai_mGetActiveCPU()` will always returns 0.

ERRORS

N/A

SEE ALSO

`ai_mGetActiveCore`

`ai_mGetActiveThread`

`ai_mSetActiveCPU`

`ai_mSetActiveCore`

`ai_mSetActiveThread`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mGetActiveThread`

Get the currently targeted thread.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetActiveThread (int mHandle, uint16_t *ActiveThread);
```

DESCRIPTION

`ai_mGetActiveThread()` retrieves the thread that is currently being targeted by the ITP driver, on the node identified by `mHandle`, and returns it in `ActiveThread`.

NOTES:

The `ActiveThread` will always return to the default value (0) after `ai_mResetUUT()`, `ai_mRunUUT()`, or, on first loading of the ITP driver library to memory.

RETURN VALUE

`ai_mGetActiveThread()` will always returns 0.

ERRORS

N/A

SEE ALSO

`ai_mGetActiveCore`

`ai_mGetActiveCPU`

`ai_mSetActiveCPU`

`ai_mSetActiveCore`

`ai_mSetActiveThread`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mGetBreakpoint`

Determine which breakpoint register met the breakpoint condition.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetBreakpoint (int mHandle, uint8_t *BreakpointNo);
```

DESCRIPTION

`ai_mGetBreakpoint()` determines which breakpoint register on the target core, on the node specified by `mHandle`, met the breakpoint condition. `BreakpointNo` returns 0, 1, 2, or 3 to indicate whether it was `BreakpointAddr0`, `BreakpointAddr1`, `BreakpointAddr2` or `BreakpointAddr3` (from `ai_mSetBreakpoint()`) respectively that met the condition).

This function should only be called if absolutely sure the target core is in debug mode. Calling the function before a breakpoint condition has been met will force the target core into debug mode (meaning the breakpoint condition will never be met).

Refer to Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A for more information on debug registers and setting breakpoints.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), ai_mSetBreakpoint() will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mGetBreakpoint()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_GET_BKPT	Error during execution of get breakpoint(s) operation
AI_INVALID_BKPT	Invalid breakpoint
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mSetBreakpoint

ai_mExitDebugMode

ai_mWaitforDebugMode

© 2021 ASSET InterTech, Inc.

NAME

`ai_mGetDebugModeStatus`

Check the debug mode status of the currently targeted core.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetDebugModeStatus (int mHandle, bool *Status);
```

DESCRIPTION

`ai_mGetDebugModeStatus()` retrieves the debug mode status for the currently targeted core on the node identified by `mHandle`. On successful completion Status will be `false` if current target is not in debug mode, otherwise `true` if it is in debug mode.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mGetDebugModeStatus()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success completion of the sequence, `ai_mGetDebugModeStatus()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP

© 2021 ASSET InterTech, Inc.

`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_STS`	Error reported during execution of get debug mode status
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mEnterDebugMode

ai_mExitDebugMode

Scan chain init: Error getting ownership of the TAP. Scan chain init: Error configuring JTAG.

Error reported during execution of get debug mode status. Error restoring JTAG/configuration registers to default.

© 2021 ASSET InterTech, Inc.

NAME

ai_mGetITPScanChainTopology

Get the current scan chain topology.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetITPScanChainTopology (int mHandle, ai_ITP_topology_t
*itpMap, bool Cstatestomin);
```

DESCRIPTION

`ai_mGetITPScanChainTopology()` interrogates and populates information on the current scan chain topology, and the status of devices in the scan chain, on the node specified by `mHandle`. The information will be returned through `itpMap`. `Cstatestomin` is used to control how C-State registers are set before the function executes. Set to `true`, C-State registers will be set to a minimum state (result is that nodes should be set to a maximum powered up state), meaning all available cores should always be powered up. Set to `false`, C-State registers retain their power up (BIOS) defaults. In this case, it is possible to encounter a broken scan chain, particularly if the target is entering very low power modes.

The `ai_ITP_topology_t` structure is made up of a number of nested sub structures. (NOTE: KNL and SNB/IVB/HSX/BDX/SKX/CLX/ICX/SPR have differing topologies, and so, some parameters within the structure are mutually exclusive, where some apply only to KNL, and others apply only to SNB/IVB/HSX/BDX/SKX/CLX/ICX/SPR).

`ai_ThreadTopology_t`

`bool activetarget`	defines if this thread is the currently targeted thread.

`ai_CoreTopology_t`

`int numthreads`	Number of threads in this core. (Normally 1 or 2 (or 4 in the case of KNL)).
`unsigned long idcode`	IDCODE of the core.
`ai_ThreadTopology_t thread[4]`	Thread topology structure for each core's threads.
`bool enabled`	Core enabled (Cores can be disabled by BIOS, or be in a low power state (C-state)). Valid only for SNB/IVB/HSX/BDX/SKX. Unused for KNL.
`int coreref`	Core number identifier within current CPU. Valid only for SNB/IVB/HSX/BDX/SKX. Unused for KNL.

© 2021 ASSET InterTech, Inc.

`ai_TileTopology_t`

`int numcores`	Number of cores in this tile.
`unsigned long idcode`	IDCODE of the tile.
`ai_Coretopology_t core[2]`	Core topology structure for each core in the tile.
`bool enabled`	Tile enabled (Tiles can be disabled by BIOS, or be in a low power C-state).
`int tileref`	Tile number identifier within current CPU.

`ai_CPUTopology_t`

`int numuncores`	Number of uncores in this CPU (normally 1).
`unsigned long uncoreidcode`	IDCODE of the uncore.
`int numcores`	Number of cores in this CPU. Unused for KNL.
`ai_CoreTopology_t core[60]`	Core topology structure for each core of this CPU. Unused for KNL.
`int numtiles`	Number of tiles in this CPU. Valid only for KNL.
`ai_TileTopology_t tile[38]`	Tile topology structure for each tile of this CPU. Valid only for KNL.
`int numgraphdevs`	Number of graphic devices in this CPU (unused).
`unsigned long graphidcode`	IDCODE of the graphics device.
`int numiiodevs`	Number of IIO devices in this CPU (unused).
`unsigned long iioidcode`	IDCODE of the IIO device.

`ai_TCKTopology_t`

`int numCPUs`	Number of CPUs found attached to this TCK domain.
`ai_CPUTopology_t CPU[4]`	CPU topology structure for each CPU on this TCK.

`ai_ITPTopology_t`

`int numtcks`	Number of TCK domains that make up the ITP.
`char tgtsysname[50]`	Character array return system target identifier.
`ai_TCKTopology_t tck[2]`	TCK topology structure for each TCK on this ITP.

NOTES:

© 2021 ASSET InterTech, Inc.

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mGetITPScanChainTopology()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

RETURN VALUE

On success, `ai_mGetITPScanChainTopology()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_GET_ITP_TOP_ERR	Error during interrogation for topology information
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

ai_mGetPriorStateInfo

Return prior state information of the currently targeted core.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mGetPriorStateInfo (int mHandle, unsigned long
*PriorStateInfo);
```

DESCRIPTION

`ai_mGetPriorStateInfo()` returns prior state information (obtained prior to the last entry to debug mode), for the currently targeted core on the node identified by `mHandle`, with the encoded results being returned through `PriorStateInfo`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mGetPriorStateInfo()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), and only if the target is in debug mode, the function will save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success completion of the sequence, `ai_mGetPriorStateInfo()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_NOT_IN_DBG_MODE`	Target not in debug mode
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFcrashdumpDiscovery`
Crashdump Discovery using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>
```

```
bool ai_mIOSFcrashdumpDiscovery(int mHandle, int CPU, uint8_t readLen,
uint8_t subopcode, uint8_t param0, uint16_t param1, uint8_t param2,
uint8_t *data, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFcrashdumpDiscovery()` reads the Crashdump Discovery information at the given `subopcode`, `param0`, `param1`, and `param2` on the given CPU on the node identified by `mHandle`. `readLen` specifies the length to read, and should be 1, 2 or 8. *`data` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `data[0]` will contain the LSB. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing this function, be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function, and also wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFcrashdumpDiscovery()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, CrashDump - Discovery

`ai_mIOSFwritePCIConfigLocal`

`ai_mIOSFreadEndpointConfig`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

ai_mIOSFcrashdumpGetFrame
Crashdump Data using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFcrashdumpGetFrame(int mHandle, int CPU, uint8_t readLen,
uint16_t param0, uint16_t param1, uint16_t param2, uint8_t* data,
uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFcrashdumpGetFrame()` reads the Crashdump Discovery information at the given `param0`, `param1`, and `param2` on the given CPU on the node identified by `mHandle`. `readLen` specifies the length to read, and should be 8 or 16. *`data` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `data[0]` will contain the LSB. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFcrashdumpGetFrame()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, CrashDump

`ai_mIOSFwritePCIConfigLocal`

`ai_mIOSFreadEndpointConfig`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

NAME

© 2021 ASSET InterTech, Inc.

ai_mIOSFreadEndpointConfig

Read PCI configuration using OOBMSM.

SYNOPSIS

```
#include <itp_iosf.h>
```

```
bool ai_mIOSFreadEndpointConfig(int mHandle, int CPU, uint8_t readLen,
uint8_t msgType, uint8_t EndPointID, uint8_t AddressType, uint8_t
segment, uint8_t bus, uint8_t dev, uint8_t fun, uint16_t offset,
uint8_t *pcireg, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFreadEndpointConfig()` reads the PCI downstream configuration register at the given `bus`, `dev`, `fun`, and `offset` on the given CPU on the node identified by `mHandle`. `pcireg` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `pcireg[0]` will contain the LSB. `cc` is the condition code returned by the hardware. For `msgType`, `EndPointID` `AddressType` see the Intel EDS for description.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFreadEndpointConfig()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdEndPointConfig()

`ai_mIOSFwritePCIConfig`

`ai_mIOSFreadConfigLocal`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

NAME

© 2021 ASSET InterTech, Inc.

ai_mIOSFreadMSR

Read MSR using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFreadMSR(int mHandle, int CPU, uint16_t MSRaddr, uint8_t
thread, uint64_t *msrreg, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFreadMSR()` reads the MSR at the given `MSRaddr` and thread on the given CPU on the node identified by `mHandle`. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFreadMSR()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdPCIConfigLocal()

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFreadPCIConfig`
Read PCI configuration, using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>
```

```
bool ai_mIOSFreadPCIConfig(int mHandle, int CPU, uint8_t bus, uint8_t
dev, uint8_t fun, uint16_t offset, uint8_t *pcireg, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFreadPCIConfig()` reads the PCI downstream configuration register at the given `bus`, `dev`, `fun`, and `offset` on the given CPU on the node identified by `mHandle`. `pcireg` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `pcireg[0]` will contain the LSB. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFreadPCIConfig()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdPCIConfig()

`ai_mIOSFwritePCIConfig()`

`ai_mIOSFreadConfigLocal()`

`ai_mIOSFTAPownership()`

`ai_mIOSFTAPinit()`

© 2021 ASSET InterTech, Inc.

NAME

ai_mIOSFreadPCIConfigLocal
Read PCI configuration, local; using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFreadPCIConfigLocal(int mHandle, int CPU, uint8_t bus,
uint8_t dev, uint8_t fun, uint16_t offset, uint8_t readLen, uint8_t
*pcireg, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFreadPCIConfigLocal()` reads the PCI local configuration register at the given `bus`, `dev`, `fun`, and `offset` on the given CPU on the node identified by `mHandle`. `readLen` specifies the length to read, and should be 1, 2 or 4. `pcireg` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `pcireg[0]` will contain the LSB. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFreadPCIConfigLocal()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdPCIConfigLocal()

`ai_mIOSFwritePCIConfigLocal`

`ai_mIOSFreadEndpointConfig`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit()`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFreadPkgConfig`

Read Package configuration, using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFreadPkgConfig(int mHandle, int CPU,uint8_t readLen,
uint8_t index, uint16_t param, uint8_t *configData, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFreadPkgConfig()` reads the package configuration register at the given `index` and `param` on the given CPU on the node identified by `mHandle`. `readLen` specifies the length to read, and should be 4. `configData` is a pointer to a byte array to receive the return value, and should be allocated by the caller. `configData[0]` will contain the LSB. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFreadPkgConfig()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdPkgConfig()

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFTAPinit`
TAP initialization for IOSF/OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFTAPinit(int mHandle);
```

DESCRIPTION

`ai_mIOSFTAPinit()` initializes the software for the IOSF/OOBMSM accesses. It should be called once after SED library initialization and before any other IOSF calls are made.

RETURN VALUE

On success, `ai_mIOSFTAPinit()` returns `true`. On error, it will return `false`.

SEE ALSO

`ai_mIOSFTAPownership`

© 2021 ASSET InterTech, Inc.

NAME

ai_mIOSFTAPownership
Take TAP ownership for OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFTAPownership(int mHandle, bool take, int TAPno);
```

DESCRIPTION

`ai_mIOSFTAPownership()` takes ownership of the TAP prior to using it for OOBMSM access. It should be called after the call to `ai_mIOSFTAPinit()` and prior to a sequence of OOBMSM commands with `take` = `true` to acquire ownership of the TAP and called again with `take` = `false` at the end of a sequence of commands. `TAPno` is number of the device on the JTAG scan chain, starting with zero. This will normally be the CPU socket.

NOTES:

N/A

RETURN VALUE

On success, `ai_mIOSFTAPownership()` returns `true`. On error, it will return `false`.

SEE ALSO

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFwritePCIConfig`
Write PCI configuration, using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>
```

```
bool ai_mIOSFwritePCIConfig(int mHandle, int CPU, uint8_t bus, uint8_t
dev, uint8_t fun, uint16_t offset, uint8_t writeLen, uint32_t newVal,
uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFwritePCIConfig()` writes the PCI downstream configuration register at the given `bus`, `dev`, `fun`, and `offset` on the given CPU on the node identified by `mHandle`. `newVal` is the new value to write to the register. `writeLen` is the length to write, and should be 1, 2, or 4. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFwritePCIConfig()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, RdPCIConfig()

`ai_mIOSFreadPCIConfig`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mIOSFwritePCIConfigLocal`
Write PCI configuration, local; using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFwritePCIConfigLocal(int mHandle, int CPU, uint8_t bus,
uint8_t dev, uint8_t fun, uint16_t offset, uint8_t writeLen, uint32_t
newVal, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFwritePCIConfigLocal()` writes the PCI local configuration register at the given `bus`, `dev`, `fun`, and `offset` on the given CPU on the node identified by `mHandle`. `writeLen` specifies the length to write, and should be 1, 2 or 4. `newVal` is the value to write. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFwritePCIConfigLocal()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, WrPCIConfigLocal()

`ai_mIOSFreadPCIConfigLocal`

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

ai_mIOSFwritePkgConfig
Write Package configuration, using OOBMSM

SYNOPSIS

```
#include <itp_iosf.h>

bool ai_mIOSFwritePkgConfig(int mHandle, int CPU, uint8_t index,
uint16_t param, uint32_t newValue, uint8_t writeLen, uint8_t *cc);
```

DESCRIPTION

`ai_mIOSFwritePkgConfig()` writes the package configuration register at the given `index` and `param` on the given CPU on the node identified by `mHandle`. `writeLen` specifies the length to written, and should be 4. `newValue` is the data to be written. `cc` is the condition code returned by the hardware.

NOTES:

Prior to executing the function be sure initialize the SED library and call the `ai_mIOSFTAPinit()` function and wrap a section of IOSF calls with calls to `ai_mIOSFTAPownership()`.

RETURN VALUE

On success, `ai_mIOSFwritePkgConfig()` returns `true`. On error, it will return `false`. Also check the condition code `cc` for any errors returned by the hardware.

SEE ALSO

External Design Specification for the processor, vol 1, PECI Functional Description, WrPkgConfig()

`ai_mIOSFTAPownership`

`ai_mIOSFTAPinit`

© 2021 ASSET InterTech, Inc.

NAME

ai_mIsPowerOn
Check if target is powered up.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mIsPowerOn (int mHandle);
```

DESCRIPTION

`ai_mIsPowerOn()` checks if the target identified by `mHandle` is powered up by evaluating the level on the HOOK0 pin of the XDP interface.

RETURN VALUE

On success, `ai_mIsPowerOn()` returns 0 (i.e. target is powered up). On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up.

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mNavigatetoTAPState`
Navigate JTAG controller to a state

SYNOPSIS

```
#include <itp_driver.h>

int ai_mNavigatetoTAPState(int mHandle, AI_tapendstate endState);
```

DESCRIPTION

`ai_mNavigatetoTAPState()` navigates the JTAG state machine from its current JTAG state to the requested JTAG end state. `mHandle` identifies the node to be executed on.

`endState` specifies the end state for the JTAG state machine to navigate to. `endState` takes one of the following arguments:

`AI_tlr`	Test-Logic-Reset.
`AI_rti`	Run-Test-Idle.

Caution should be exercised when using `ai_mNavigatetoTAPState()` as it may cause undefined/erratic behavior in other ITP Driver functions.

RETURN VALUE

On success, `ai_mNavigatetoTAPState()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_VCC_NOT_FOUND`	Target not powered up.
`AI_ERR_TAP_NAVIGATE`	ITP Driver controller problem executing TAP navigate.

SEE ALSO

`ai_mScanDr`

`ai_mScanIr`

© 2021 ASSET InterTech, Inc.

NAME

ai_mOpen

Initialize communication channel with ITP Driver FPGA controller device.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mOpen (AI_pdcselector PdcNo, BOOL QuietAttach, int *mHandle);
```

DESCRIPTION

`ai_mOpen()` initializes the ITP Driver FPGA controller device for use and initializes the required ITP Driver support structures and classes.

The function opens up a new mapping to the ITP Driver controller device/IP block specified by `PdcNo`. `PdcNo` refers to the logical node number, which the function then uses to map to a LOC device. A number of different types of LOC devices are now supported (new LOC types typically being added with new processor family support). The search order for the different type of LOC devices is KLOC, QLOC, LOC.

`AI_pdc_0` locfpga0 or QLOC1/KLOC1-physical node 1.

`AI_pdc_1` locfpga1 or QLOC0/KLOC0-physical node 1.

`AI_pdc_2` locfpga2 or QLOC0/KLOC0-physical node 0.

`AI_pdc_3` locfpga3 or QLOC1/KLOC1-physical node 0.

`QuietAttach` is a dummy parameter (for now!).

On successful completion of the function, `mHandle` will provide a unique identifier 'connection' to the specified `PdcNo`. This parameter should be used when calling any other subsequent ITP driver 'ai_m' functions.

The first call with a specific `PdcNo` to `ai_mOpen()` / `ai_mOpenEx()` (after first instantiation), will result in a self-test on the ITP Driver controller device logic. Subsequent calls to `ai_mOpen()` / `ai_mOpenEx()` (with the same `PdcNo`, where the driver is still memory-resident) will bypass the self-test section. `ai_mOpen()` / `ai_mOpenEx()` must be called before calling any functions that utilize the ITP Driver controller device.

RETURN VALUE

On success, `ai_mOpen()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

`AI_DRIVER_ALREADY_OPEN`	Driver already open
`AI_INVALID_PARAM`	Invalid parameter
`AI_ERR_OPEN_L0CFPGA`	Can’t find specified device
`AI_ERR_PDC_MMAP`	Problem mapping
`AI_ERR_INITJTAG_EIE`	Error initializing the locfpga device JTAG controller
`AI_ERR_TGT_CLASS`	Error initializing required processor class object

SEE ALSO

`ai_mOpenEx`

`ai_mClose`

© 2021 ASSET InterTech, Inc.

NAME

ai_mOpenEx

Initialize communication channel with ITP Driver controller device - an alternative futureproof function.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mOpenEx (const char *devname, unsigned int PctOffset, unsigned
int ir_mask_num, AI_pdcse- lector PdcNo, int *mHandle);
```

DESCRIPTION

ai_mOpenEx() provides a method (an alternative to ai_mOpen) for initialization of the ITP Driver controller device and initialization of the required ITP Driver support structures and classes.

The function opens up a path with the ITP Driver FPGA controller specified by devname (e.g. /dev/qlocf-pga1), and proceeds to set up a new mapping to the device/IP block specified by PctOffset.

ir_mask_num specifies the PCT interrupt mask register to be associated with the opened device (for use with ITP breakpoints, etc..).. ir_mask_num can take on the following values:

```
0      IREQ
1      USRA  (LOC->PDC/QLOC->QPDC/HLOC->HPDC/KLOC->KPDC physical node 0)
2      USRB
3      USRA1 (QLOC->QPDC/HLOC->HPDC/KLOC->KPDC physical node 1)
4      USRB1
```

The input PdcNo provides a unique logical node number to be associated with the combined set of devname, PctOffset and ir_mask_num parameters. Accepted values are AI_pdc_0, AI_pdc_1, AI_pdc_2 or AI_pdc_3.

On successful completion of the function, mHandle will provide a unique identifier 'connection' to the specified PdcNo. This parameter should be used when calling any other subsequent ITP driver 'ai_m' functions.

The first call with a specific PdcNo to ai_mOpen()/ai_mOpenEx() (after first instantiation), will result in a self-test on the ITP Driver controller device logic. Subsequent calls to ai_mOpen()/ ai_mOpenEx() (with the same PdcNo, where the driver is still memory-resident) will bypass the self-test section. ai_mOpen()/ai_mOpenEx() must be called before calling any functions that utilize the ITP Driver controller device.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mOpenEx()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_DRIVER_ALREADY_OPEN`	Driver already open
`AI_ERR_OPEN_DEV`	Problem opening device
`AI_ERR_PDC_MMAP`	Problem mapping
`AI_ERR_INITJTAG_EIE`	Error initializing the locfpga device JTAG controller
`AI_ERR_TGT_CLASS`	Error initializing required processor class object

SEE ALSO

`ai_mOpen`

`ai_mClose`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mRamBusTest`
Execute a RAM Bus Test diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRamBusTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, char* ErrorString);
```

DESCRIPTION

`ai_mRamBusTest()` executes a test to diagnose the data and address buses between the CPU and a RAM area.

`mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

`ai_mRamBusTest()` assumes a 64-bit data bus width (for data bus and byte enable lane testing). If an error is diagnosed, ErrorString will return the diagnostic information.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mRamBusTest()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

`ai_mRamBusTest()` is divided in to 4 sub-tests, executed in sequence.

1. Data bus hi/lo test.

© 2021 ASSET InterTech, Inc.

2. Data bus shorts test.
3. Byte enables test.
4. Address bus test.

If any sub-test fails the diagnostic, it returns immediately, skipping execution of any subsequent sub-tests.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mRamBusTest()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up.
AI_MAX_UNCORES_EXCEEDED	Scan chain init: Maximum possible UNCORES exceeded.
AI_INVALID_CORE_IDCODE_DET	Scan chain init: An invalid core IDCODE was detected.
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: An invalid Uncore IDCODE was detected.
AI_ERR_TAP_OWNERSHIP	Scan chain init: Error getting ownership of the TAP.
AI_CFG_JTAG	Scan chain init: Error configuring JTAG.
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address.
AI_RAM_BUS_DATA_HILO	Data bus hi/lo failure diagnosed.
AI_RAM_BUS_DATA_SHORT	Data bus shorts failure diagnosed.
AI_RAM_BUS_BYTE_ENABLES	Byte enables lane(s) failure diagnosed.
AI_RAM_BUS_ADDRESS	Address bus failure diagnosed.
AI_RAM_TEST_HALTED_USR	Execution of RAM Bus Test interrupted by user.
AI_DBG_MODE_RAM_TEST	Error during execution of Boot RAM Bus Test.
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG/Configuration settings to default.

SEE ALSO

ai_mRamBusTestChannel

ai_mRamBusTestviaFIFO

ai_mBasicRWRamBusTest

© 2021 ASSET InterTech, Inc.

ai_mRamBusTest

`ai_mRWRamTest`

`ai_mDRamRefreshTest`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mRamBusTestChannel`

Execute a RAM Bus Test Channel diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRamBusTestChannel (int mHandle, uint64_t StartAddress,
uint64_t EndAddress, AI_buswidth BusWidth, uint64_t ChannelSize,
AI_nooframchannels NoOfChannels, char* ErrorString);
```

DESCRIPTION

`ai_mRamBusTestChannel()` executes a test to diagnose the data and address buses between the CPU and a RAM area.

`mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

`BusWidth` takes on one of the following values to specify the data bus width for the data and byte enable lanes testing.

8	8-bit operation width
16	16-bit operation width
32	32-bit operation width
64	64-bit operation width

`ChannelSize` and `NoOfChannels` are used to form a mask for the address bus test algorithm.

`NoOfChannels` takes on one of the following values:

`AI_one`	Single-channel RAM
`AI_two`	2-channel RAM
`AI_four`	4-channel RAM
`AI_eight`	8-channel RAM

If an error is diagnosed, `ErrorString` will return the diagnostic information.

`ai_mRamBusTestChannel()` is functionally similar to `ai_mRamBusTest()`, except that the user can specify the data bus width, and the algorithm to do address bus testing differs.

© 2021 ASSET InterTech, Inc.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), ai_mRamBusTestChannel() will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

`ai_mRamBusTestChannel()` is divided in to 4 sub-tests, executed in sequence.

1. Data bus hi/lo test.
2. Data bus shorts test.
3. Byte enables test.
4. Address bus channel test.

If any sub-test fails the diagnostic returns immediately, skipping execution of any subsequent sub-tests.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mRamBusTestChannel()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_VCC_NOT_FOUND`	Target not powered up.
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: Maximum possible UNCORES exceeded.
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: An invalid core IDCODE was detected.
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: An invalid Uncore IDCODE was detected.

© 2021 ASSET InterTech, Inc.

AI_ERR_TAP_OWNERSHIP	Scan chain init: Error getting ownership of the TAP.
AI_CFG_JTAG	Scan chain init: Error configuring JTAG.
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address.
AI_RAM_BUS_DATA_HILO	Data bus hi/lo failure diagnosed.
AI_RAM_BUS_DATA_SHORT	Data bus shorts failure diagnosed.
AI_RAM_BUS_BYTE_ENABLES	Byte enables lane(s) failure diagnosed.
AI_RAM_BUS_ADDRESS	Address bus failure diagnosed.
AI_RAM_TEST_HALTED_USR	Execution of RAM Bus Test interrupted by user.
AI_DBG_MODE_RAM_TEST	Error during execution of Boot RAM Bus Test.
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG/Configuration settings to default.

SEE ALSO

ai_mRamBusTest

ai_mRamBusTestviaFIFO

ai_mBasicRWRamBusTest

ai_mRWRamTest

ai_mDRamRefreshTest

© 2021 ASSET InterTech, Inc.

NAME

ai_mRamBusTestviaFIFO
Execute a RAM Bus Test via FIFO diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRamBusTestviaFIFO (int mHandle, uint64_t StartAddress,
uint64_t EndAddress, uint64_t FIFOLimit, char* ErrorString);
```

DESCRIPTION

`ai_mRamBusTestviaFIFO()` executes a test to diagnose the data and address buses between the CPU and a RAM area.

`mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

`FIFOLimit` specifies the depth of FIFO for data bus testing.

If an error is diagnosed, `ErrorString` will return the diagnostic information.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mRamBusTestviaFIFO()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

`ai_mRamBusTestviaFIFO()` is divided in to 3 sub-tests, executed in sequence.

© 2021 ASSET InterTech, Inc.

1. Data bus integrity test.
2. Byte enables test.
3. Address bus channel test.

If any sub-test fails the diagnostic returns immediately, skipping execution of any subsequent sub-tests.

For CPUs that have built-in FIFOs, `FIFOLimit` informs the data bus testing algorithm the amount of transactions required to propagate data on to the actual data bus, so that the data bus can be properly diagnosed.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mRamBusTestviaFIFO()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up.
AI_MAX_UNCORES_EXCEEDED	Scan chain init: Maximum possible UNCORES exceeded.
AI_INVALID_CORE_IDCODE_DET	Scan chain init: An invalid core IDCODE was detected.
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: An invalid Uncore IDCODE was detected.
AI_ERR_TAP_OWNERSHIP	Scan chain init: Error getting ownership of the TAP.
AI_CFG_JTAG	Scan chain init: Error configuring JTAG.
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address.
AI_RAM_BUS_VIA_FIFO_DATA	Data bus hi/lo failure diagnosed.
AI_RAM_BUS_VIA_FIFO_BYTE_ENABLES	Byte enables lane(s) failure diagnosed.
AI_RAM_BUS_VIA_FIFO_ADDRESS	Address bus failure diagnosed.
AI_RAM_TEST_HALTED_USR	Execution of RAM Bus Test interrupted by user.
AI_DBG_MODE_RAM_TEST	Error during execution of Boot RAM Bus Test.
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG/Configuration settings to default.

SEE ALSO

ai_mRamBusTest

ai_mRamBusTestChannel

© 2021 ASSET InterTech, Inc.

ai_mRamBusTestviaFIFO

`ai_mBasicRWRamBusTest`

`ai_mRWRamTest`

`ai_mDRamRefreshTest`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReadCR`

Read from a CR register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadCR (int mHandle, ai_crregister crreg, uint64_t
*RegisterData);
```

DESCRIPTION

`ai_mReadCR()` retrieves RegisterData from crreg on the currently targeted core, on the node identified by `mHandle`.

`crreg` specifies the CR register to be read. `crreg` takes one of the following arguments;

`AI_CR0`	CR0
`AI_CR2`	CR2
`AI_CR3`	CR3
`AI_CR4`	CR4
`AI_CR8`	CR8

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), ai_mReadCR() will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mReadCR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_CR`	Error during execution of read from CR operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteCR`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReadCSR`
Read from a CSR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadCSR (int mHandle, uint16_t DeviceNo, uint16_t FunctionNo,
uint16_t Offset, uint32_t

*RegisterData);
```

DESCRIPTION

`ai_mReadCSR()` retrieves `RegisterData` from the targeted CSR on the currently targeted CPU, on the node identified by `mHandle`.

The targeted CSR is made up by combining the `DeviceNo`, `FunctionNo` and `Offset` fields to form the CSR register address.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadCSR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

`ai_mReadCSR()` submits instructions via a non-debug mode related JTAG access mechanism. It does not require the target CPU to be in debug mode to operate successfully.

This non-debug mode related JTAG access mechanism only exists on **Nehalem** targets. Attempts to use this function on other targets will result in a 'Function not supported' error. See the IOSF routines for more current JTAG access mechanisms that do not require a halt.

RETURN VALUE

On success, `ai_mReadCSR()` returns 0. On error, it will return one of the following values:

ERRORS

© 2021 ASSET InterTech, Inc.

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_READ_CSR	Error during execution of read from CSR operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mWriteCSR

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReadDescriptorTableRegister`
Read from a descriptor table register.

SYNOPSIS

```
#include <itp_driver.h>
```

```
int ai_mReadDescriptorTableRegister (int mHandle, ai_dtrregister
dtrreg, uint64_t *Base, uint64_t *Limit, uint64_t *Selector, uint64_t
*Attributes);
```

DESCRIPTION

`ai_mReadDescriptorTableRegister()` retrieves the descriptor table register fields `Base`, `Limit`, `Selector`, `Attributes` from the descriptor table register specified by `dtrreg` on the currently targeted core, on the node identified by `mHandle`.

`dtrreg` takes one of the following arguments:

`AI_GDTR`	GDTR
`AI_LDTR`	LDTR
`AI_IDTR`	IDTR
`AI_TR`	TR

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), ai_mReadDescriptorTableRegis- ter() will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mReadDescriptorTableRegister()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_DTRREG`	Error during execution of read from descriptor table
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteDescriptorTableRegister`

© 2021 ASSET InterTech, Inc.

NAME

ai_mReadDR
Read from a debug register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadDR (int mHandle, ai_drregister drreg, uint64_t
*RegisterData);
```

DESCRIPTION

`ai_mReadDR()` retrieves `RegisterData` from drreg on the currently targeted core, on the node identified by `mHandle`.

`drreg` specifies the DR register to be read. `drreg` takes one of the following arguments:

`AI_DR0`	DR0
`AI_DR1`	DR1
`AI_DR2`	DR2
`AI_DR3`	DR3
`AI_DR6`	DR6
`AI_DR7`	DR7

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadDR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action,

© 2021 ASSET InterTech, Inc.

which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mReadDR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_DR`	Error during execution of read from DR operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteDR`

© 2021 ASSET InterTech, Inc.

NAME

ai_mReadGPR
Read from a GPR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadGPR (int mHandle, AI_gprregister GprNo, uint64_t
*RegisterData);
```

DESCRIPTION

`ai_mReadGPR()` retrieves `RegisterData` from `GprNo` on the currently targeted core, on the node identified by `mHandle`.

If `GprNo` is within the RAX - R15 range, then `RegisterData` will be read directly from the register on the target core. Otherwise `RegisterData` will be read from the target core register space in the 'processor state buffer'.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadGPR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mReadGPR()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_GPR`	Error during execution of read from GPR operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteGPR`

© 2021 ASSET InterTech, Inc.

NAME

ai_mReadIO

Read from an I/O location.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadIO (int mHandle, uint16_t IoAddress, void *IoData,
ai_mBuswidth BusWidth);
```

DESCRIPTION

`ai_mReadIO()` submits instruction(s) to the target core to read the data at the I/O location specified in `IoAddress`, on the node identified by `mHandle`. The data read will be returned through `IoData`. The `BusWidth` argument defines the width of operation to be carried out.

`AI_bwio8`	8-bit operation
`AI_bwio16`	16-bit operation
`AI_bwio32`	32-bit operation

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadIO()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

© 2021 ASSET InterTech, Inc.

On success, `ai_mReadIO()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_READ_IO	Error during execution of read from IO operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mWriteIO

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReadMemory`

Read from a memory location.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadMemory (int mHandle, uint64_t MemoryAddress, void
*MemoryData, AI_buswidth

BusWidth);
```

DESCRIPTION

`ai_mReadMemory()` submits instruction(s) to the target core to read the data at the memory location speci-fied in `MemoryAddress`, on the node identified by `mHandle`. The data read will be returned through `MemoryData`.

The `BusWidth` argument defines the width of operation to be carried out.

`AI_bwmem8`	8-bit operation
`AI_bwmem16`	16-bit operation
`AI_bwmem32`	32-bit operation
`AI_bwmem64`	64-bit operation

Normal debug mode operations are performed with the target core in 32-bit operating mode. If `MemoryAddress` is greater than 0xffffffff, or `BusWidth` equals `AI_bwmem64` then the function will switch the target core to 64-bit mode, prior to execution of the read operation. On completion of the operation, a flag will be set to return the target core back to 32-bit on the next debug operation (if the next operation does not require 64-bit mode).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadMemory()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in

© 2021 ASSET InterTech, Inc.

debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mReadMemory()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_READ_MEM	Error during execution of read from memory operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteMemory`

© 2021 ASSET InterTech, Inc.

NAME

ai_mReadMSR
Read from an MSR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadMSR (int mHandle, uint64_t MsrAddress, uint64_t
*RegisterData);
```

DESCRIPTION

`ai_mReadMSR()` retrieves `RegisterData` from `MsrAddress` on the currently targeted core, on the node identified by `mHandle`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadMSR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mReadMSR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up

© 2021 ASSET InterTech, Inc.

`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_MSR`	Error during execution of read from MSR operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteMSR`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReadSegmentRegister`
Read from a segment register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReadSegmentRegister (int mHandle, ai_segmentregister segreg,
uint64_t *Base, uint64_t *Limit, uint64_t *Selector, uint64_t
*Attributes);
```

DESCRIPTION

`ai_mReadSegmentRegister()` retrieves the segment register fields `Base`, `Limit`, `Selector`, `Attributes` from the segment register specified by `segreg` on the currently targeted core, on the node identified by `mHandle`.

`segreg` takes one of the following arguments:

AI_CS	CS
AI_DS	DS
AI_SS	SS
AI_ES	ES
AI_FS	FS
AI_GS	GS

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReadSegmentRegister()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used

© 2021 ASSET InterTech, Inc.

for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mReadSegmentRegister()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_READ_SEGREG`	Error during execution of read from segment register operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteSegmentRegister`

© 2021 ASSET InterTech, Inc.

NAME

ai_mResetDetect
Set/check target reset detection circuitry.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mResetDetect (int mHandle, AI_resetdetectoption mode);
```

DESCRIPTION

`ai_mResetDetect()` gives options to be able to clear and detect resets that may occur on the target, on the node specified by `mHandle`.

`mode` defines the operation to be carried out:

`AI_enable_clear`	Enable and clear reset detection circuit
`AI_disable`	Disable reset detection circuit
`AI_check`	Check if a reset has been detected

NOTES:

It has been noticed that external agents cause problems for the ITP driver. One of these problems is the occurrence of resets driven by external agents (i.e. other than the reset driven by `ai_mResetUUT()` or `ai_mRunUUT()` functions). The occurrence of such resets generally cause undefined behavior with the ITP driver library. `ai_mResetDetect()` allows the ability to detect if such a reset occurred.

RETURN VALUE

On success, `ai_mResetDetect()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_RESET_DETECTED`	Reset detected

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mResetUUT`
Apply a DBR reset to the connected target(s) and hold them in debug mode.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mResetUUT (int mHandle);
```

DESCRIPTION

`ai_mResetUUT()` applies a reset pulse to the connected target(s) on the node identified by `mHandle` via the DBR (HOOK7) line on the ITP Driver FPGA controller device. The length of reset pulse applied to DBR can be adjusted using ResetPulseDuration via `ai_mConfig()`. Before applying the pulse to the DBR line, the ITP FGPA controller is primed to halt the CPU core(s) on detection of activity on the CPU RESET (HOOK6) signal. (i.e. a ' debug mode at reset condition' will be trapped). Any subsequent activity on the CPU RESET signal will also trap a ' debug mode at reset condtion'. However, execution of ai_mClose() will negate the ability to halt the CPU core(s) on the occurrence of a CPU RESET.

Debug mode status is not checked on completion of the reset operation. (i.e. the function does not check that all connected CPU core(s) have entered debug mode).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mResetUUT()` will first of all perform a power check on the target.

BUGS

Intel have reported a silicon bug with some versions of SandyBridge, where the CPU cores do not remain in debug mode thru and after reset. The recommended workaround is to apply a follow-on pulse to the PREQ (OBSFN_n0) line(s) shortly after RESET (HOOK6) de-assertion. Although this workaround will also be applied to unaffected CPUs, it will have no adverse effect for them.

RETURN VALUE

On success completion of the sequence, `ai_mResetUUT()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mRunUUT

© 2021 ASSET InterTech, Inc.

NAME

ai_mReturnIDCode
Return the TAP IDCODE.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReturnIDCode (int mHandle, uint32_t *IDCode);
```

DESCRIPTION

ai_mReturnIDCode() retrieves the TAP IDCODE from the currently targeted core on the node identified by mHandle and returns it in IDCode.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the PowerCheck option is not disabled via ai_mConfig()), ai_mReturnIDCode() will first of all perform a power check on the target.

Also, by default (if the ScanChainSetup option is not disabled via ai_mConfig()), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success, ai_mReturnIDCode() returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_IDCODE	Error retrieving IDCODE

© 2021 ASSET InterTech, Inc.

`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReturnIDCodewithOverscan`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReturnIDCodewithOverscan`
Return the TAP IDCODE while executing and checking an overscan pattern.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReturnIDCodewithOverscan (int mHandle, uint32_t *IDCode);
```

DESCRIPTION

`ai_mReturnIDCodewithOverscan()` retrieves the TAP IDCODE from the currently targeted core on the node identified by `mHandle` and returns it in `IDCode`. While executing the scan an overscan pattern will be attached to the TDI for the DR scan. The overscan pattern received at TDO is checked to ensure it matches the input pattern.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReturnIDCodewithOverscan()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success, `ai_mReturnIDCodewithOverscan()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP

© 2021 ASSET InterTech, Inc.

AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_ERR_OVERSCAN_FAIL	TDO Overscan data does not match expected data
AI_IDCODE	Error retrieving IDCODE
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mReturnIDCode

© 2021 ASSET InterTech, Inc.

NAME

`ai_mReturnSiliconID`
Return the silicon ID values.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mReturnSiliconID (int mHandle, uint32_t *RepoID, uint32_t
*DieCfg);
```

DESCRIPTION

`ai_mReturnSiliconID()` retrieves the silicon ID parameters from the currently targeted CPU on the node identified by `mHandle`. The data read will be returned through `RepoID` and `DieCfg`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mReturnSiliconID()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On success, `ai_mReturnSiliconID()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG

© 2021 ASSET InterTech, Inc.

`AI_IDCODE`	Error retrieving IDCODE
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default
`AI_SILICONID_NOT_IMPLEMENTED`	Function not supported (for the target CPU type)

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

ai_mRomCrcTest

Execute a CRC test on a memory block/range.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRomCrcTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, uint16_t* CrcValue);
```

DESCRIPTION

`ai_mRomCrcTest()` submits instruction(s) to the target core to execute the 'CRC computation' machine code routine on the target core, on the node identified by `mHandle`, to compute of a memory block starting at `StartAddress` and ending at `EndAddress`. The computed value is returned on successful completion in CrcValue.

During execution of the 'CRC computation' machine code routine the target core will exit debug mode. The function will not return until the core has re-entered debug mode. Upon completion, the target core should immediately re-enter debug mode, at which point, the target core can then service ITP driver functions normally again.

Should the user wish to force re-entry to debug mode during execution of the 'CRC computation' machine code routine, he/she can do so by calling `ai_mStopTest()` via a forked child process. `ai_mStopTest()` will force debug mode re-entry, which will cause `ai_mRomCrcTest()` to subsequently return.

No other ITP Driver functions (other than `ai_mStopTest()`) should be called while `ai_mRomCrcTest()` is running. Because the target core is not in debug mode during user diagnostic execution, execution of other ITP Driver functions can force the target core to re-enter debug mode. In this case, `ai_mRomCrcTest()` behavior is undefined.

The 'CRC computation' machine code routine forms part of a collection of machine code routines which the ITP driver can execute. Since the routines are machine code, they must be downloaded and run from an area of memory accessible by the target core. `UUTDiagsHexFile` from `ai_mConfig()` provides the ITP driver library with a pointer to the machine code file, and `UUTDiagsBaseAddress`, also from `ai_mConfig()`, defines the memory base address from which the machine code will be run. Prior to calling the 'CRC computation' machine code routine, the function will check if the machine code routines exist at `UUTDiagsBaseAddress`, and if not, will proceed to download the file pointed to by `UUTDiagsHexFile`.

NOTES:

© 2021 ASSET InterTech, Inc.

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mRomCrcTest()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The machine code routines operate in 32-bit mode only, therefore any machine code routines will only operate in the bottom 4G memory space (i.e 0x0-0xFFFFFFFF). Behavior is undefined if the range specified extends outside this area.

When calling `ai_mRomCrcTest()`, the user should ensure that memory range specified does not overlap into the `UUTDiagsBaseAddress` memory area reserved for execution of the machine code routines. Also, the user should ensure that memory has been initialized sufficiently to allow the machine code routines to run properly. In both cases, the function may fail to return normally (i.e. unless forced using `ai_mStopTest()`).

RETURN VALUE

On success, `ai_mRomCrcTest()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_ROM_CRC	Error during execution of CRC computation operation

© 2021 ASSET InterTech, Inc.

AI_ROM_CRC_HALTED_USR	Execution of CRC computation operation halted by user
AI_ROM_CRC_HALTED_UNKNWN_SRC	Execution of CRC computation operation halted by unknown source
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address
AI_NO_EXEC_HALT_STATE	Target core indicates a HALT state – unable to execute user diagnostic
AI_NO_EXEC_WAIT_FOR_SIPI_STATE	Target core indicates a WAIT FOR SIPI state – unable to execute diag
AI_NO_EXEC_SHUTDOWN_STATE	Target core indicates a SHUTDOWN state – unable to execute diag
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

FILES

`Pentcode.hex` - machine code routines collection. Can be installed to any directory. `UUTDiagsHexFile` from `ai_mConfig()` provides the ITP driver library with a pointer to the machine code file.

SEE ALSO

`ai_mStopTest`

© 2021 ASSET InterTech, Inc.

NAME

ai_mRunUUT

Apply a DBR reset to the connected target(s) and allow them to boot.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRunUUT (int mHandle);
```

DESCRIPTION

`ai_mRunUUT()` applies a reset pulse to the connected target(s) on the node identified by `mHandle` via the DBR (HOOK7) line on the ITP Driver FPGA controller device. The length of reset pulse applied to DBR can be adjusted using ResetPulseDuration via `ai_mConfig()`. Before applying the pulse to the DBR line, the ITP controller is primed to allow the CPU core(s) free run on de-assertion of the CPU RESET (HOOK6) signal. Any subsequent activity on the CPU RESET signal will also allow the CPU cores to free run. (i.e. debug mode will not be entered).

Debug mode status is not checked on completion of the reset operation. (i.e. the function does not check that all connected CPU core(s) are running).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mRunUUT()` will first of all perform a power check on the target.

RETURN VALUE

On success completion of the sequence, `ai_mRunUUT()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up.
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG/configuration registers to default.

SEE ALSO

`ai_mResetUUT`

© 2021 ASSET InterTech, Inc.

NAME

ai_mRWRamTest
Execute a R/W RAM Test diagnostic.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mRWRamTest (int mHandle, uint64_t StartAddress, uint64_t
EndAddress, char* ErrorString);
```

DESCRIPTION

`ai_mRWRamTest()` executes a test to find and diagnose cell problems on a RAM area. `mHandle` identifies the node to execute on.

`StartAddress` specifies the start address of the range and `EndAddress` specifies the end address of the range, within which, operations will be carried out to perform the diagnostic algorithm(s).

If an error is diagnosed, `ErrorString` will return the diagnostic information.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mRWRamTest()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

The `ai_mRWRamTest()` function uses machine code routines that operate in 32-bit mode only. Therefore this diagnostic can only operate on the bottom 4G memory space (i.e 0x0-0xFFFFFFFF).

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_mRWRamTest()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter.
AI_VCC_NOT_FOUND	Target not powered up.
AI_MAX_UNCORES_EXCEEDED	Scan chain init: Maximum possible UNCORES exceeded.
AI_INVALID_CORE_IDCODE_DET	Scan chain init: An invalid core IDCODE was detected.
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: An invalid Uncore IDCODE was detected.
AI_ERR_TAP_OWNERSHIP	Scan chain init: Error getting ownership of the TAP.
AI_CFG_JTAG	Scan chain init: Error configuring JTAG.
AI_SA_GREATER_THAN_EA	Start address cannot be greater than end address.
AI_RW_RAM	Write RAM failure diagnosed.
AI_RAM_TEST_HALTED_USR	Execution of RAM Bus Test interrupted by user.
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function.
AI_FILE_LOAD_ERR	Error loading hex file (required for test)
AI_DBG_MODE_RAM_TEST	Error during execution of Boot RAM Bus Test.
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG/Configuration settings to default.

SEE ALSO

ai_mRamBusTest

ai_mRamBusTestChannel

ai_mRamBusTestviaFIFO

ai_mBasicRWRamTest

ai_mDRamRefreshTest

NAME

ai_mScanDr

Execute a JTAG DR Scan sequence

© 2021 ASSET InterTech, Inc.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mScanDr(int mHandle, uint32_t count, uint32_t *tdi, uint32_t
*tdo, AI_tapendstate endState);
```

DESCRIPTION

`ai_mScanDr()` executes a JTAG DR scan sequence on the JTAG lines attached to current target (ITP Driver controller device).

`mHandle` identifies the node to be executed on.

`count` specifies the no of DR bits to be transmitted.

`tdi` is a pointer to the input data. If `tdi` is NULL then binary 1' s will be used as the input data.

`tdo` is a pointer for the output data. If `tdo` is NULL then the function assumes no output data is to be returned.

`endState` specifies the end state for the JTAG state machine will return to on completion of the scan. `endState` takes one of the following arguments;

`AI_tlr`	Test-Logic-Reset
`AI_rti`	Run-Test-Idle
`AI_pausedr`	Pause-DR
`AI_pauseir`	Pause-IR

Caution should be exercised when using `ai_mScanDr()` as it may cause undefined/erratic behavior of other ITP Driver functions.

RETURN VALUE

On success, `ai_mScanDr()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_VCC_NOT_FOUND`	Target not powered up.
`AI_ERR_SCANDR`	ITP Driver controller problem executing DR scan.

© 2021 ASSET InterTech, Inc.

ai_mScanDr

SEE ALSO

`ai_mScanIr`

© 2021 ASSET InterTech, Inc.

NAME

ai_mScanIr
Execute a JTAG IR Scan sequence

SYNOPSIS

```
#include <itp_driver.h>

int ai_mScanIr(int mHandle, uint32_t count, uint32_t *tdi, uint32_t
*tdo, AI_tapendstate endState);
```

DESCRIPTION

`ai_mScanIr()` executes a JTAG IR scan sequence on the JTAG bus attached to current target (ITP Driver controller device).

`mHandle` identifies the node to be executed on.

`count` specifies the no of IR bits to be transmitted.

`tdi` is a pointer to the input data. If `tdi` is NULL, then binary 1' s will be used as the input data.

`tdo` is a pointer for the output data. If `tdo` is NULL, then the function assumes no output data is to be returned.

`endState` specifies the end state for the JTAG state machine will return to on completion of the scan. `endState` takes one of the following arguments;

`AI_tlr`	Test-Logic-Reset
`AI_rti`	Run-Test-Idle
`AI_pausedr`	Pause-DR
`AI_pauseir`	Pause-IR

Caution should be exercised when using `ai_mScanIr()` as it may cause undefined/erratic behavior in other ITP Driver functions.

RETURN VALUE

On success, `ai_mScanIr()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.

© 2021 ASSET InterTech, Inc.

`AI_VCC_NOT_FOUND`	Target not powered up.
`AI_ERR_SCANIR`	ITP Driver controller problem executing IR scan.

SEE ALSO

`ai_mScanDr`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetActiveCore`

Set the currently targeted core.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetActiveCore (int mHandle, uint16_t ActiveCore);
```

DESCRIPTION

`ai_mSetActiveCore()` sets `ActiveCore` as the core that is to be targeted by the ITP driver, on the node identified by `mHandle`.

Valid values for `ActiveCore` are 1 thru 76.

NOTES:

The active core will always return to the default value (1) after `ai_mResetUUT()`, `ai_mRunUUT()`, or on first loading of the ITP driver library to memory.

RETURN VALUE

On success, `ai_mSetActiveCore()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_TGT_CORE_INVALID`	Invalid Target core selection (Active CPU/Core/Thread returned to default value).

SEE ALSO

`ai_mGetActiveCore`

`ai_mGetActiveCPU`

`ai_mGetActiveThread`

`ai_mSetActiveCPU`

`ai_mSetActiveThread`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetActiveCPU`

Set the currently targeted CPU.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetActiveCPU (int mHandle, uint16_t ActiveCPU);
```

DESCRIPTION

`ai_mSetActiveCPU()` sets ActiveCPU as the CPU that is to be targeted by the ITP driver, on the node identified by `mHandle`.

Valid values for `ActiveCPU` are 1 thru 4.

NOTES:

The active CPU will always return to the default value (1) after `ai_mResetUUT()`, `ai_mRunUUT()`, or on first loading of the ITP driver library to memory.

RETURN VALUE

On success, `ai_mSetActiveCPU()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_TGT_CORE_INVALID`	Invalid Target core selection (Active CPU/Core/Thread returned to default value).

SEE ALSO

`ai_mGetActiveCore`

`ai_mGetActiveCPU`

`ai_mGetActiveThread`

`ai_mSetActiveCore`

© 2021 ASSET InterTech, Inc.

ai_mSetActiveCPU

`ai_mSetActiveThread`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetActiveThread`
Set the currently targeted thread.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetActiveThread (int mHandle, uint16_t ActiveThread);
```

DESCRIPTION

`ai_mSetActiveThread()` sets ActiveThread as the thread that is to be targeted by the ITP driver, on the node identified by `mHandle`.

Valid values for ActiveThread are 0 thru 3.

NOTES:

The active thread will always return to the default value (1) after `ai_mResetUUT()` , `ai_mRunUUT()`, or, on first loading of the ITP driver library to memory.

RETURN VALUE

On success, `ai_mSetActiveThread()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter.
`AI_TGT_CORE_INVALID`	Invalid Target core selection (Active CPU/Core/Thread returned to default value).

SEE ALSO

`ai_mGetActiveCore`

`ai_mGetActiveCPU`

`ai_mGetActiveThread`

`ai_mSetActiveCore`

`ai_mSetActiveCPU`

© 2021 ASSET InterTech, Inc.

NAME

ai_mSetBreakpoint

Set up breakpoint(s).

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetBreakpoint (int mHandle, uint64_t BreakpointAddr0,
AI_Breakpointtype BreakpointType0, uint64_t BreakpointAddr1,
AI_Breakpointtype BreakpointType1, uint64_t BreakpointAddr2,
AI_Breakpointtype BreakpointType2, uint64_t BreakpointAddr3,
AI_Breakpointtype BreakpointType3);
```

DESCRIPTION

ai_mSetBreakpoint() sets up the associated target core 'processor state buffer' registers with breakpoint address registers as specified by BreakpointAddr0, BreakpointAddr1, BreakpointAddr2 and BreakpointAddr3 and the breakpoint types as specified by BreakpointType0, BreakpointType1, BreakpointType2 and BreakpointType3.

mHandle identifies the node for the operation to be executed on.

The BreakpointTypen argument defines the type of breakpoint to be set on the target core.

AI_none	No breakpoint.
AI_instr_addr	Break on instruction execution only.
AI_data_write	Break on data writes only.
AI_io_read_write	Break on I/O read or writes.
AI_data_read_write	Break on data reads or writes but not instruction fetches.

Execution of ai_mSetBreakpoint() will disable single step mode if previously set up by ai_mSetRunMode(). However, ai_mSetBreakpoint() and ai_mSetRunMode() can still be used in sequence to debug suspect areas of code.

Refer to Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A for more information on debug registers and setting breakpoints.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the PowerCheck option is not disabled via ai_mConfig()), ai_mSetBreakpoint() will first of all perform a power check on the target.

© 2021 ASSET InterTech, Inc.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

Breakpoints can only be set on the bottom 4G memory space (i.e. 0x0-0xFFFFFFFF).

RETURN VALUE

On success, `ai_mSetBreakpoint()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_SET_BKPT	Error during execution of set up breakpoint(s) operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mGetBreakpoint

ai_mExitDebugMode

ai_mWaitforDebugMode

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetDebugModeCheckFlag`

Set the debug mode check flag.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetDebugModeCheckFlag (int mHandle, bool EnableDMCheck);
```

DESCRIPTION

`ai_SetDebugModeCheckFlag()` sets `EnableDMCheck` to indicate to the ITP driver whether debug mode should be checked during ITP driver function operation, for the node specified by `mHandle`. On first loading the ITP driver library, `EnableDMCheck` takes on a default value of `true`.

RETURN VALUE

`ai_SetDebugModeCheckFlag()` always returns 0.

ERRORS

N/A

© 2021 ASSET InterTech, Inc.

NAME

ai_mSetinitbreak

Set the init break flag.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetinitbreak (int mHandle, bool initbreak);
```

DESCRIPTION

ai_mSetinitbreak() sets the initbreak flag, for the node identified by mHandle.

initbreak specifies how the init break flag internal to the target is to be set. Set to true, the target will redirect to debug mode after all state initialization, but before branching to the reset vector. Set to false, there will be no redirect to debug mode after all state initialization.

NOTES:

The actual update from the flag to target processor/core register(s) is linked to the SaveModifyArch global parameter being set to true. Only when this parameter is true, and an ITP driver library function that executes/requires the SaveModifyArch procedure, will the registers get updated. (i.e. the register(s) will only get updated by ITP driver library function(s) that require the target processor to be in debug mode).

By default, when the ITP driver library is first loaded into memory, initbreak is set to false. Also, any use of ai_mSetTargetCPUType to change the target CPU to a new type will reset initbreak to false.

RETURN VALUE

On success, ai_mSetinitbreak() returns 0. On error, it will return one of the following values:

ERRORS

N/A

SEE ALSO

ai_mSetmachinecheckbreak

ai_mSetsmmentrybreak

© 2021 ASSET InterTech, Inc.

ai_mSetinitbreak

`ai_mSetshutdownbreak`

`ai_mConfig`

© 2021 ASSET InterTech, Inc.

NAME

ai_mSetmachinecheckbreak

Set the machine check break flag.

SYNOPSIS

```
#include <itp_driver.h>

int ai_metmachinecheckbreak (int mHandle, bool machinecheckbreak);
```

DESCRIPTION

`ai_mSetmachinecheckbreak()` sets the `machinecheckbreak` flag, for the node identified by `mHandle`.

`machinecheckbreak` specifies how the machine check break flag internal to the target is to be set. Set to `true`, the target will redirect to debug mode before the exception handler occurs. Set to `false`, there will be no redirect to debug mode before exception handling.

NOTES:

The actual update from the flag to target processor/core register(s) is linked to the `SaveModifyArch` global parameter being set to `true`. Only when this parameter is `true`, and an ITP driver library function that executes/requires the `SaveModifyArch` procedure, will the registers get updated. (i.e. the register(s) will only get updated by ITP driver library function(s) that require the target processor to be in debug mode).

By default, when the ITP driver library is first loaded into memory, `machinecheckbreak` is set to `false`. Also, any use of `ai_mSetTargetCPUType` to change the target CPU to a new type will reset `machinecheckbreak` to `false`.

RETURN VALUE

On success, `ai_mSetmachinecheckbreak()` returns 0. On error, it will return one of the following values:

ERRORS

N/A

SEE ALSO

© 2021 ASSET InterTech, Inc.

ai_mSetmachinecheckbreak

`ai_mSetinitbreak`

`ai_mSetsmmentrybreak`

`ai_mSetshutdownbreak`

`ai_mConfig`

© 2021 ASSET InterTech, Inc.

NAME

ai_mSetRunMode

Set the target execution mode.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetRunMode (int mHandle, AI_runmode RunMode);
```

DESCRIPTION

ai_mSetRunMode() changes the target core/CPU execution mode behavior, on the node specified by mHandle, when it is next set to run again (i.e. when debug mode is next exited).

If RunMode is equal to AI_run, when debug mode is next exited, the target will start/restart processing its instruction queue. Otherwise, if RunMode is equal to AI_step, when debug mode is next exited, the target will only process the next instruction in its queue, and then immediately re-enter debug mode again (i.e. a single-step).

Any breakpoint conditions previously set up by ai_mSetBreakpoint() will still remain valid if RunMode is set to AI_run.

Refer to the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A for more information on debug registers and single-step mode.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the PowerCheck option is not disabled via ai_mConfig()), ai_mSetRunMode() will first of all perform a power check on the target.

Also, by default (if the ScanChainSetup option is not disabled via ai_mConfig()), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the SaveModifyArch option is not disabled via ai_mConfig()), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mSetRunMode()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_SET_RUN_MODE	Error during execution of set run mode operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mExitDebugMode`

`ai_mWaitforDebugMode`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetshutdownbreak`
Set the shutdown break flag.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetshutdownbreak (int mHandle, bool shutdownbreak);
```

DESCRIPTION

`ai_mSetshutdownbreak()` sets the `shutdownbreak` flag, for the node identified by `mHandle`.

`shutdownbreak` specifies how the shutdown break flag internal to the target is to be set. Set to `true`, the target will redirect to debug mode just prior to issuing a special bus cycle, and disabling processing. Set to `false`, there will be no redirect to debug mode prior to issuing the special bus cycle.

NOTES:

The actual update from the flag to target processor/core register(s) is linked to the `SaveModifyArch` global parameter being set to `true`. Only when this parameter is `true`, and an ITP driver library function that executes/requires the `SaveModifyArch` procedure, will the registers get updated. (i.e. the register(s) will only get updated by ITP driver library function(s) that require the target processor to be in debug mode).

By default, when the ITP driver library is first loaded into memory, `shutdownbreak` is set to `false`. Also, any use of `ai_mSetTargetCPUType` to change the target CPU to a new type will reset `shutdownbreak` to `false`.

RETURN VALUE

On success, `ai_mSetshutdownbreak()` returns 0. On error, it will return one of the following values:

ERRORS

N/A

SEE ALSO

`ai_mSetinitbreak`

© 2021 ASSET InterTech, Inc.

ai_mSetshutdownbreak

`ai_mSetmachinecheckbreak`

`ai_mSetsmmentrybreak`

`ai_mConfig`

© 2021 ASSET InterTech, Inc.

NAME

ai_mSetsmmentrybreak

Set the SMM entry break flag.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetsmmentrybreak (int mHandle, bool smmentrybreak);
```

DESCRIPTION

ai_mSetsmmentrybreak() sets the smmentrybreak flag, for the node identified by mHandle

smmentrybreak specifies how the SMM entry break flag internal to the target is to be set. Set to true, the target will redirect to debug mode after completing an SMI handler macro operation. Set to false, there will be no redirect to debug mode after the SMI macro handler.

NOTES:

The actual update from the flag to target processor/core register(s) is linked to the SaveModifyArch global parameter being set to true. Only when this parameter is true, and an ITP driver library function that executes/requires the SaveModifyArch procedure, will the registers get updated. (i.e. the register(s) will only get updated by ITP driver library function(s) that require the target processor to be in debug mode).

By default, when the ITP driver library is first loaded into memory, smmentrybreak is set to false. Also, any use of ai_mSetTargetCPUType to change the target CPU to a new type will reset smmentrybreak to false.

RETURN VALUE

On success, ai_mSetsmmentrybreak() returns 0. On error, it will return one of the following values:

ERRORS

N/A

SEE ALSO

ai_mSetinitbreak

© 2021 ASSET InterTech, Inc.

ai_mSetsmmentrybreak

`ai_mSetmachinecheckbreak`

`ai_mSetshutdownbreak`

`ai_mConfig`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetTap`
Select the target TAP.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetTap (int mHandle, int tap);
```

DESCRIPTION

`ai_mSetTap()` informs the ITP driver of the target TAP selection, on the node specified by `mHandle`.

`tap` can take on the following values:

-1	No TAP
0	TAP 0
1	TAP 1
2	TAP 2
3	TAP 3
4	TAP 4
5	Internal controller TAP
6	Reserved for internal use only

RETURN VALUE

On success, `ai_mSetTap()` returns 0.

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mSetTargetCPUType`

Select the target CPU.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mSetTargetCPUType (int mHandle, AI_CPUtype CPUtype);
```

DESCRIPTION

`ai_mSetTargetCPUType()` informs the ITP driver of the target CPU selection, on the node specified by `mHandle`.

`CPUtype` can take on the following values:

`AI_nehalem`	Nehalem target.
`AI_sandybridge`	Sandybridge/Ivybridge/Haswell/Broadwell/Skylake/Cascade Lake/Ice Lake/Sapphire Rapids target. (ITP Driver can transparently determine between Sandybridge, Ivybridge, Haswell, Broadwell, Skylake, Ice Lake, and Sapphire Rapids targets. Note that the SED library can, as of Haswell and beyond, automatically detect the type of processor; thus this is a legacy API and should always be invoked with `AI_sandybridge` on current platforms).

Default value (on ITP driver library load) is `AI_nehalem`.

RETURN VALUE

On success, `ai_mSetTargetCPUType()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_ERR_TGT_CLASS`	Error instantiating target class

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

ai_mSetTargetCPUType

© 2021 ASSET InterTech, Inc.

NAME

`ai_mStopTest`
Stop/Interrupt a currently running test.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mStopTest (int mHandle);
```

DESCRIPTION

`ai_mStopTest()` informs the ITP Driver to cease any currently running diagnostic routines on the node identified by `mHandle` (applies to ITP driver functions that operate on a range mainly).

NOTES:

`ai_mStopTest()` should be invoked via a child process, to halt/return and force any required cleanup of any functions currently being run by the parent process. `ai_mStopTest()` only supplies notification to stop the parent process function. Its return does not indicate the parent process's function completion. Wait for the parent process function to return to indicate completion.

RETURN VALUE

`ai_mStopTest()` always returns 0.

SEE ALSO

`ai_mFillMemory`

`ai_mCheckMemory`

`ai_mRomCrcTest`

`ai_mBootRomBusTest`

`ai_mRamBusTest`

`ai_mRamBusTestChannel`

`ai_mRamBusTestViaFifo`

`ai_mBasicR-WRamTest`

`ai_mRWRamTest`

`ai_mDRamRefreshTest`

© 2021 ASSET InterTech, Inc.

NAME

ai_muregraw

Execute a uregraw operation.

SYNOPSIS

```
#include <itp_ureg_raw.h>

int ai_muregraw (int mHandle, uint32_t device, uint32_t portID, char*
registerType, uint32_t bar, uint32_t deviceNumber, uint32_t function,
uint32_t address, uint32_t iascope, uint32_t* wrValue, uint32_t*
rdValue);
```

DESCRIPTION

ai_muregraw() executes a uregraw operation on the targeted CPU, on the node specified by mHandle. This function is provided mainly as an interface for Intel CScripts.

device	An uncore, core, or thread device.
portID	uArch parameter that specifies the message channel destination port.
registerType	A string specifying the register type. Can be one of 'cr', 'cfg'.
bar	Base Access Register (BAR) parameter.
deviceNumber	Register device attribute. If registerType = 'cr', then deviceNumber is treated as a core ID.
function	Register function attribute. If registerType = 'cr', then function is treated as a thread ID.
address	16-bit address or offset to access.
iaScope	The IA scope parameter (typically set to 0).
wrValue	Specifies a 32-bit value to write to the register. Set to NULL if doing a read.
rdValue	Used to return 32-bit read value. Set to NULL if doing a write.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the PowerCheck option is not disabled via ai_mConfig()), ai_muregraw() will first of all perform a power check on the target.

© 2021 ASSET InterTech, Inc.

Also, by default (if the ScanChainSetup option is not disabled via ai_mConfig()), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

RETURN VALUE

On successful completion of the diagnostic with no errors, ai_muregraw() returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_UREG_RAW_NOT_IMPLEMENTED	Not a valid function (IVYBRIDGE/IVYTOWN ONLY supports)
AI_UREG_RAW_INV_PARAM	Invalid parameter for uregraw
AI_UREG_RAW_MAX_POLL_TIMEOUT	Timeout during uregraw operation
AI_UREG_RAW_EXEC_ERR	uregraw execution error
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mEnableoxmdebug

ai_muregraw64

NAME

ai_muregraw64

Execute a uregraw operation.

© 2021 ASSET InterTech, Inc.

SYNOPSIS

```
#include <itp_ureg_raw.h>

int ai_muregraw64 (int mHandle, uint32_t device, uint32_t portID,
char* registerType, uint32_t bar, uint32_t deviceNumber, uint32_t
function, uint32_t address, uint32_t iascope, uint64_t* wrValue,
uint64_t* rdValue);
```

DESCRIPTION

`ai_muregraw64()` executes a `uregraw` operation on the targeted CPU, on the node specified by `mHandle`. This function is provided mainly as an interface for Intel CScripts.

`device`	An uncore, core, or thread device.
`portID`	uArch parameter that specifies the message channel destination port.
`registerType`	A string specifying the register type. Can be one of 'cr', 'cfg'.
`bar`	Base Access Register (BAR) parameter.
`deviceNumber`	Register device attribute. If registerType = 'cr', then deviceNumber is treated as a core ID.
`function`	Register function attribute. If registerType = 'cr', then function is treated as a thread ID.
`address`	16-bit address or offset to access.
`iaScope`	The IA scope parameter (typically set to 0).
`wrValue`	Specifies a 64-bit value to write to the register. Set to NULL if doing a read.
`rdValue`	Used to return 64-bit read value. Set to NULL if doing a write.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_muregraw64()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

The scan chain will be returned to its original state on function completion.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On successful completion of the diagnostic with no errors, `ai_muregraw64()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_UREG_RAW_NOT_IMPLEMENTED	Not a valid function (IVYBRIDGE/IVYTOWN ONLY supports)
AI_UREG_RAW_INV_PARAM	Invalid parameter for uregraw
AI_UREG_RAW_MAX_POLL_TIMEOUT	Timeout during uregraw operation
AI_UREG_RAW_EXEC_ERR	uregraw execution error
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mEnableoxmdebug

ai_muregraw

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWaitforDebugMode`
Wait for debug mode re-entry.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWaitforDebugMode (int mHandle);
```

DESCRIPTION

`ai_mWaitforDebugMode()` waits for the targeted core on the node identified by `mHandle` to re-enter debug mode. This function would typically be used to capture when a single step or breakpoint condition has been met. During execution, the process is suspended until a PRDY interrupt condition occurs. Upon this, the currently targeted core's debug mode status is checked. If in debug mode the function returns, and if the target core is not in debug mode the function loops and re-enters the suspended state again.

Should the user wish to force re-entry to debug mode during execution of the 'wait for debug mode' routine, he/she can do so by calling `ai_mStopTest()` via a forked child process. `ai_mStopTest()` will force debug mode re-entry, which will cause `ai_mWaitforDebugMode()` to subsequently return.

No other ITP Driver functions (other than `ai_mStopTest()`) should be called while `ai_mWaitforDebugMode()` is running. Because the target core is not in debug mode, execution of other ITP Driver functions can force the target core to re-enter debug mode. In this case, `ai_mWaitforDebugMode()` behavior is undefined.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWaitforDebugMode()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

RETURN VALUE

On success completion of the sequence, `ai_mWaitforDebugMode()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WAIT_DM	Error during execution wait for debug mode operation
AI_WAIT_DM_HALTED_USR	Execution of wait for debug mode interrupted by user
AI_ERR_PCT_INT_IOCTL	Error reported from IOCTL() function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mStopTest

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWBINVD`

Execute the WBINVD instruction.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWBINVD (int mHandle);
```

DESCRIPTION

`ai_mWBINVD()` executes the `WBINVD` instruction on the currently targeted core, on the node identified by `mHandle`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWBINVD()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWBINVD()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded

© 2021 ASSET InterTech, Inc.

`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_INVD`	Error during execution of WBINVD operation
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

N/A

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWriteCR`

Write to a CR register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteCR (int mHandle, ai_crregister crreg, uint64_t
RegisterData);
```

DESCRIPTION

`ai_mWriteCR()` writes `RegisterData` to `crreg` of the currently targeted core, on the node identified by `mHandle`.

`crreg` specifies the CR register to be written. `crreg` takes one of the following arguments:

AI_CR0	CR0
AI_CR2	CR2
AI_CR3	CR3
AI_CR4	CR4
AI_CR8	CR8

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteCR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mWriteCR()` returns 0. On error, it will return one of the following values:

ERRORS

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_CR	Error during execution of write to CR operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadCR`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWriteCSR`
Write to a CSR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteCSR (int mHandle, uint16_t DeviceNo, uint16_t FunctionNo,
uint16_t Offset, uint32_t RegisterData);
```

DESCRIPTION

`ai_mWriteCSR()` writes `RegisterData` to the targeted CSR on the currently targeted CPU, on the node identified by `mHandle`.

The targeted CSR is made up by combining the `DeviceNo`, `FunctionNo` and `Offset` fields to form the CSR register address.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteCSR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

`ai_mWriteCSR()` submits instructions via a non-debug mode related JTAG access mechanism. It does not require the target CPU to be in debug mode to operate successfully.

The non-debug mode related JTAG access mechanism only exists on Nehalem targets. Attempts to use this function on other targets will result in a 'Function not supported' error.

RETURN VALUE

On success, `ai_mWriteCSR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up

© 2021 ASSET InterTech, Inc.

`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_WRITE_CSR`	Error during execution of write to CSR function
`AI_LIB_FUNC_NOT_SUPPORTED`	Function not supported (only Nehalem supports)
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadCSR`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWriteDescriptorTableRegister`
Write to a descriptor table register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteDescriptorTableRegister (int mHandle, ai_dtrregister dtrreg, uint64_t *Base, uint64_t *Limit, uint64_t *Selector, uint64_t *Attributes);
```

DESCRIPTION

`ai_mWriteDescriptorTableRegister()` writes the descriptor table register fields `Base`, `Limit`, `Selector`, `Attributes` to the descriptor table register specified by `dtrreg` of the currently targeted core, on the node identified by `mHandle`.

`dtrreg` takes one of the following arguments:

`AI_GDTR`	`GDTR`
`AI_LDTR`	`LDTR`
`AI_IDTR`	`IDTR`
`AI_TR`	`TR`

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteDescriptorTableRegister()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mWriteDescriptorTableRegister()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_DTRREG	Error during execution of write to descriptor table operation
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadDescriptorTableRegister`

© 2021 ASSET InterTech, Inc.

NAME

ai_mWriteDR
Write to a DR register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteDR (int mHandle, ai_drregister drreg, uint64_t
RegisterData);
```

DESCRIPTION

`ai_mWriteDR()` writes `RegisterData` to `drreg` of the currently targeted core, on the node identified by `mHandle`.

`drreg` specifies the CR register to be written. `drreg` takes one of the following arguments:

AI_DR0	DR0
AI_DR1	DR1
AI_DR2	DR2
AI_DR3	DR3
AI_DR6	DR6
AI_DR7	DR7

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteDR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action,

© 2021 ASSET InterTech, Inc.

which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWriteDR()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_DR	Error during execution of write to DR function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadDR`

© 2021 ASSET InterTech, Inc.

NAME

ai_mWriteGPR
Write to a GPR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteGPR (int mHandle, AI_gprregister GprNo, uint64_t
RegisterData);
```

DESCRIPTION

`ai_mWriteGPR()` writes `RegisterData` to `GprNo` of the currently targeted core, on the node identified by `mHandle`.

If `GprNo` is within the RAX - R15 range, then `RegisterData` will be written directly to the register on the target core. Otherwise `RegisterData` will be written to the target core register space in the 'processor state buffer' held.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteGPR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWriteGPR()` returns 0. On error, it will return one of the following values:

© 2021 ASSET InterTech, Inc.

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_WRITE_GPR	Error during execution of write to GPR function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mReadGPR

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWriteIO`

Write to an I/O location.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteIO (int mHandle, uint16_t IoAddress, void *IoData,
ai_mBuswidth BusWidth);
```

DESCRIPTION

`ai_mWriteIO()` submits instruction(s) to the target core to write data specified in `IoData` to the I/O location specified in `IoAddress`, on the node identified by `mHandle`.

The `BusWidth` argument defines the width of operation to be carried out.

`AI_bwio8`	`8-bit operation`
`AI_bwio16`	`16-bit operation`
`AI_bwio32`	`32-bit operation`

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteIO()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

© 2021 ASSET InterTech, Inc.

RETURN VALUE

On success, `ai_mWriteIO()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_IO	Error during execution of write to IO function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mWriteIO`

© 2021 ASSET InterTech, Inc.

NAME

`ai_mWriteMemory`

Write to a memory location.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteMemory (int mHandle, uint64_t MemoryAddress, void
*MemoryData, ai_mBuswidth BusWidth);
```

DESCRIPTION

`ai_mWriteMemory()` submits instruction(s) to the target core to write data specified in `MemoryData` to the memory location specified in `MemoryAddress`, on the node identified by `mHandle`. The `BusWidth` argument defines the width of operation to be carried out.

`AI_bwmem8`	`8-bit operation`
`AI_bwmem16`	`16-bit operation`
`AI_bwmem32`	`32-bit operation`
`AI_bwmem64`	`64-bit operation`

Normal debug mode operations are performed with the target core in 32-bit operating mode. If `MemoryAddress` is greater than 0xffffffff, or `BusWidth` equals `AI_bwmem64`, the function will switch the target core to 64-bit mode prior to execution of the write operation. On completion of the operation, a flag will be set to return the target core back to 32-bit on the next debug operation (if the next operation does not require 64-bit mode).

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteMemory()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used

© 2021 ASSET InterTech, Inc.

for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWriteMemory()` returns 0. On error, it will return one of the following values:

ERRORS

AI_INVALID_PARAM	Invalid parameter
AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_MEM	Error during execution of write to memory function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadMemory`

© 2021 ASSET InterTech, Inc.

NAME

ai_mWriteMSR

Write to an MSR.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteMSR (int mHandle, uint64_t MsrAddress, uint64_t
RegisterData);
```

DESCRIPTION

`ai_mWriteMSR()` writes `RegisterData` to `MsrAddress` of the currently targeted core, on the node identified by `mHandle`.

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteMSR()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWriteMSR()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter

© 2021 ASSET InterTech, Inc.

AI_VCC_NOT_FOUND	Target not powered up
AI_MAX_UNCORES_EXCEEDED	Scan chain init: maximum possible Uncores exceeded
AI_INVALID_CORE_IDCODE_DET	Scan chain init: an invalid Core IDCODE was detected
AI_INVALID_UNCORE_IDCODE_DET	Scan chain init: an invalid Uncore IDCODE was detected
AI_ERR_TAP_OWNERSHIP	Scan chain init: error getting ownership of the TAP
AI_CFG_JTAG	Scan chain init: error configuring JTAG
AI_DBG_MODE_WRITE_MSR	Error during execution of write to MSR function
AI_RESTORE_JTAG_CFG_DEFAULT	Error restoring JTAG configuration settings to default

SEE ALSO

ai_mReadMSR

© 2021 ASSET InterTech, Inc.

NAME

ai_mWriteSegmentRegister

Write to a segment register.

SYNOPSIS

```
#include <itp_driver.h>

int ai_mWriteSegmentRegister (int mHandle, ai_segmentregister segreg,
uint64_t *Base, uint64_t *Limit, uint64_t *Selector, uint64_t
*Attributes);
```

DESCRIPTION

`ai_mWriteSegmentRegister()` writes the segment register fields `Base`, `Limit`, `Selector`, and `Attributes` to the segment register specified by `segreg` of the currently targeted core, on the node identified by `mHandle`.

`segreg` takes one of the following arguments:

AI_CS	CS
AI_DS	DS
AI_SS	SS
AI_ES	ES
AI_FS	FS
AI_GS	GS

NOTES:

Prior to executing the function, the following actions may be carried out:

By default (if the `PowerCheck` option is not disabled via `ai_mConfig()`), `ai_mWriteSegmentRegister()` will first of all perform a power check on the target.

Also, by default (if the `ScanChainSetup` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to interrogate/bring up and ensure all target devices (cores) are alive on the scan chain.

Lastly, by default (if the `SaveModifyArch` option is not disabled via `ai_mConfig()`), the function will perform necessary actions to force all connected devices (cores) in to debug mode (if not already in debug mode), and save the architectural state of the target core (again, if not already saved previously) to a 'processor state buffer'. The 'processor state buffer' is a storage area in host memory used to temporarily store the architectural state registers of the core, such that these registers can then be used

© 2021 ASSET InterTech, Inc.

for other debug operations. The action to save the 'processor state buffer' is a one-time only action, which only requires execution/re-execution if the target is reset, the core 'processor state buffer' was previously restored, or a new instance of the driver (.so) is loaded.

RETURN VALUE

On success, `ai_mWriteSegmentRegister()` returns 0. On error, it will return one of the following values:

ERRORS

`AI_INVALID_PARAM`	Invalid parameter
`AI_VCC_NOT_FOUND`	Target not powered up
`AI_MAX_UNCORES_EXCEEDED`	Scan chain init: maximum possible Uncores exceeded
`AI_INVALID_CORE_IDCODE_DET`	Scan chain init: an invalid Core IDCODE was detected
`AI_INVALID_UNCORE_IDCODE_DET`	Scan chain init: an invalid Uncore IDCODE was detected
`AI_ERR_TAP_OWNERSHIP`	Scan chain init: error getting ownership of the TAP
`AI_CFG_JTAG`	Scan chain init: error configuring JTAG
`AI_DBG_MODE_WRITE_SEGREG`	Error during execution of write to segment register function
`AI_RESTORE_JTAG_CFG_DEFAULT`	Error restoring JTAG configuration settings to default

SEE ALSO

`ai_mReadSegmentRegister`

© 2021 ASSET InterTech, Inc.