

**Previsé™**

Technical Manual

# **Bailey DCS Simulator API**

March 2017

Prepared by:



**Previsive Inc.®**  
Managing Critical Microsoft® Technology

### Printed in Canada

This document is subject to continuous improvement, and as such is subject to change without notice. Feedback or inquiries regarding this document are welcome. Contact Previsive via [www.previsive.com](http://www.previsive.com).

---

**Proprietary Notice:** This document and related software contain proprietary information which represents trade secrets of Previsive Inc. and may not be copied or disclosed, except as provided in the license with Previsive Inc. Use of the information in this document and related software, for the reverse engineering of OPsCon, or for the development or manufacture of similar software is prohibited. The information in this document is subject to change without notice and should not be construed as a commitment by Previsive Inc. Previsive Inc. assumes no responsibility for any errors that may be in this document.

Copyright © 1996 - 2017 Previsive Inc. All rights reserved.

Unauthorized reproduction is a violation of Previsive Inc. copyright.

### Trademarks

Previsive and OPsCon are trademarks or registered trademarks of Previsive Inc.

Bailey, Net 90, INFI 90, Harmony and others are registered trademarks of ABB.

Microsoft, Windows and others are trademarks or registered trademarks of the Microsoft Corporation.

All other brand or product names are trademarks or registered trademarks of their respective holders.

### Notice

Previsive Inc., its partners, affiliates, employees, and agents, and the authors of, and contributors to, this publication and the software it represents, specifically disclaim all liabilities and warranties, express and implied (including warranties of merchantability and fitness for a particular purpose), for the accuracy, currency, completeness, and/or reliability of the information contained herein, and/or for the fitness for any particular use, and/or for the performance of any material, and/or for equipment selected in whole or part by the user in reliance upon information contained herein. Selection of materials and/or equipment is at the sole risk of the user of this publication.

# Table of Contents

<b>1</b>	<b>Bailey DCS Simulator API .....</b>	<b>1</b>
1.1	Purpose of API .....	1
1.2	Microsoft Visual C++ .....	1
1.3	Intended Audience.....	1
1.4	Sample Application.....	1
<b>2</b>	<b>API Programmers Reference .....</b>	<b>2</b>
2.1	Interface Object: CDcsControl .....	2
2.2	Overview of Methods.....	2
	Error Message at USB License Key Issue .....	2
	Error Message at Bad API Method Call .....	2
2.3	Bailey DCS Simulator Control Functions.....	3
	Method: Restart.....	3
	Method: Pause.....	4
	Method: Continue.....	5
	Method: SaveCfgFiles .....	5
	Method: Throttle.....	6
2.4	Date and Time Functions .....	7
	Method: DateTime.....	7
2.5	Process Data Read & Write Methods.....	9
	Typical Control Logic.....	9
	Function Blocks.....	10
	A Typical Function Block .....	10
	Process Host Writes to Block Address.....	11
	Process Host Reads from Block .....	12
	Input/Output Block Types .....	12
	Method: ReadBlockArray.....	13
	Method: WriteBlockArray.....	16
	Data Structures .....	16
2.6	DCS State File Functions.....	21
	Overview.....	21
	What is Saved to, and Restored from, State File.....	21
	The DCS process state file .....	22
	Path for DCS State Files .....	22
	Method: DcsState.....	22
	Performance Notes .....	23
2.7	Command Log and Replay Functions .....	24
	Command Buffer Details .....	24
	Method: CommandLog operations.....	25
<b>3</b>	<b>Usage Scenarios .....</b>	<b>28</b>
3.1	Simulator Startup .....	28
3.2	Simulator Shutdown.....	28
3.3	Normal Quiescent Simulator Operation.....	28
3.4	Command Logging.....	29
3.5	Backtrack.....	29



# 1 Bailey DCS Simulator API

This manual provides an Application Programming Interface (API) Manual for the Previsé Bailey DCS Simulator.

---

## 1.1 Purpose of API

The primary purpose of this API is to support connection of external process modeling software to the Previsé Bailey DCS Simulator.

---

## 1.2 Microsoft Visual C++

This interface manual has been developed to define all API functions in terms of the Microsoft Visual C++ language.

---

**NOTE:** This interface will also support connection via C, other C++ variants, and via Visual Basic. Contact Previsé if you require interface details for languages other than Microsoft Visual C++.

---

---

## 1.3 Intended Audience

This manual is intended for use by knowledgeable programming staff that have a need to connect an external software product to the Previsé Bailey DCS Simulator. The most likely reason that this would be done is to support connection of external process modeling software to the Bailey DCS Simulator.

It is assumed that the reader is:

- An experienced C++ software developer
- Familiar with DCS IO concepts in general
- Familiar with the Bailey DCS Simulator function.

---

## 1.4 Sample Application

Ask Previsé about a sample software application, written in Visual C++, to make use of this API. This application, with full source code available, is intended to simplify the process of developing software to use at this interface.

## 2 API Programmers Reference

### 2.1 Interface Object: CDcsControl

CDcsControl is intended to be used with C/C++ applications, but also can be used with VB applications. This interface created as an automation interface.

### 2.2 Overview of Methods

Method	Purpose
<b>Simulator Control Methods</b>	
Restart	Restart all modules currently configured into the DCS (via the <i>default.INI</i> file).
Pause	Pause all the modules currently configured into the DCS (via the <i>default.INI</i> file).
Continue	Continue (Resume) execution of paused modules.
Throttle	Change speed of simulation with respect to real time from 0.1 X to 10 X real time.
SaveCFGFiles	Save controller module binary CFG file overwriting CFG files already present, for any controller module CFG file configured into the DCS (via the <i>default.INI</i> file).
<b>Date Time Methods</b>	
DateTime	Read or write system date and time for computer hosting Bailey DCS Simulator
<b>Process Data Read &amp; Write Methods</b>	
ReadBlockArray	Read an array of block values.
WriteBlockArray	Write an array of block values.
<b>Simulator Process State File Methods</b>	
DcsState	Provides functions to: <ul style="list-style-type: none"> <li>▪ Save DCS process state to a state file</li> <li>▪ Restore a DCS state file</li> <li>▪ Retrieve a list of all stored DCS state files</li> <li>▪ Delete a specific DCS state file</li> </ul>
<b>Operator Command Log and Replay Methods</b>	
CommandLog	Supports logging of operator commands via operator console including: <ul style="list-style-type: none"> <li>▪ Start command log. Writes incoming operator commands to retrieval buffer.</li> <li>▪ Get buffer of time tagged operator commands logged to buffer.</li> <li>▪ Reset (empty) operator command log buffer.</li> <li>▪ Stop operator command logging.</li> <li>▪ Replay single operator command</li> </ul>

#### Error Message at USB License Key Issue

If the USB license key is not installed, or if the API is not Enabled on the installed USB license key, then the API is disabled. In this event the error message received by the calling application will be:

*pvError: DCS\_CONTROL\_API\_DISABLED(0x12).*

#### Error Message at Bad API Method Call

Error response at bad API call that can not be interpreted is:

*pvError: DCS\_CONTROL\_REQUEST\_BAD\_DATA (0x11)*

## 2.3 Bailey DCS Simulator Control Functions

### Method: Restart

Restarts all the modules currently configured in the DCS. Causes read of default.INI file and causes all configured CFG files to be loaded prior to start. Equivalent to STOP and START of Bailey DCS Simulator service. This function will also:

- Clear Command Log Buffer.
- Clear any Forced Value settings (these may be restored via RestoreState function).

### Syntax:

```
HRESULT Restart(VARIANT *pvError)
```

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2 or VT\_I4

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error, where

DCS\_CONTROL\_NO\_ERROR is a constant defined in DCSIO.H header value, 0 is numeric value. It is recommended to use a constants defined in DCSIO.H file rather than a numeric value.

DCS\_CONTROL\_RESTART\_FAILED(3)- restart failed

### C/C++ Code example:

```
HRESULT hr;
VARIANT var;
ICDcsControl *m_pDcs = NULL;
hr = CoCreateInstance(CLSID_CDcsControl, NULL, CLSCTX_SERVER, IID_
ICDcsControl, (LPVOID FAR *) &m_pDcs)
if( hr == S_OK)
{
    hr = m_pDcs->Restart(&var);
    UINT iError;
    if( var.vt == VT_I2 ) iError = var.iVal;
    else ( var.vt == VT_I4 ) iError =(UINT) var.lVal;
    else; //error
    m_pDcs->Release();
}
```

## Method: Pause

Pause execution of all the modules configured in the DCS. Causes the execution of each segment of each module to stop execution once the current execution is completed. In other words, if a segment is in mid-execution then that segment will complete it's current pass before this method returns.

### Syntax:

HRESULT Pause(VARIANT \*pvError)

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_PAUSED(1) - execution has already been paused

### Performance Note

Once this method call is made no new segment execution passes will be started. This function will not return until all module execution already in progress is stopped (i.e. each segment completes it's current execution pass).

The nominal time that it will take to return is calculated as follows:

- $T_{\text{return}} = T_{\text{segment}} \times \text{Load}_{\text{CPU}}$
- Where:
  - $T_{\text{return}}$  – A time which should be greater than the time it takes to return.
  - $T_{\text{segment}}$  – The average segment execution interval (typical 250 millisecc)
  - $\text{Load}_{\text{CPU}}$  - The average CPU load at 1X real time (%)
- Example:
  - $T_{\text{segment}}$  – 250 millisecc
  - $\text{Load}_{\text{CPU}}$  – 4%
  - $T_{\text{return}}$  – 250 milliseconds X 4% = 10 milliseconds



## Method: Continue

Continue execution all the modules currently configured in the DCS, using the current state within the DCS as the start state. Modules are not reloaded.

### Syntax:

```
HRESULT Continue(VARIANT *pvError)
```

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_RUNNING (2)- execution has not been paused

### Performance Note

This method should return in less than 10 milliseconds.

## Method: SaveCfgFiles

This method will explicitly save all CFG files that are defined in the *default.INI* file and that are loaded into the Bailey DCS Simulator, overwriting the CFG files that are already present at the defined path.

### Syntax:

```
HRESULT SaveCfgFiles(VARIANT *pvError)
```

### Error Returns:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Error Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_CFGSAVE\_FAILED (4)- failed

## Method: Throttle

This method supports two operations:

Get – retrieves the current throttle setting.

Set – Changes the current throttle setting.

The throttle provides the means to changes the speed of simulation with respect to real time and supports execution from 0.1X to 10X real time.

---

**NOTE:** Execution at **N** times real time speeds requires that CPU load will increase by N times the real time load. The CPU must be sized appropriately to support the increased load. The Bailey DCS Simulator may become unstable if CPU load higher than 100% is needed to execute at the desired speed multiplier.

---

Default throttle at startup is 1.0.

This method will accept throttle of ANY real number between 0.1 and 10.0. However, it is recommended that actual throttle be restricted to the following domain of values:

0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0

### Syntax:

```
HRESULT Throttle(BSTR szOperation, double *pdScale, VARIANT *pvError);
```

### Where:

BSTR szOperation – operation. Values:

DCS\_CONTROL\_COMMAND\_THROTTLE\_SET (“Set”) – sets speed of simulation

DCS\_CONTROL\_COMMAND\_THROTTLE\_GET (“Get”) – retrieves speed of simulation

double \*pdScale – simulation speed (0.1 -10)

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_INVALID\_THROTTLE(0xB) – invalid throttle simulation speed(\*pdScale)

## 2.4 Date and Time Functions

These time commands are used to support date and time synchronization between the computer hosting the Bailey DCS Simulator and a process simulation host computer.

The DateTime method supports two operations:

- Get – return current simulator time (from Microsoft OS)
- Set – set current simulator time (to Microsoft OS)

### Method: DateTime

#### Syntax:

```
HRESULT DateTime(BSTR szOperation,VARIANT *pvData,VARIANT *pvError)
```

#### Where:

BSTR szOperation – operation. Values:

DCS\_CONTROL\_COMMAND\_DATETIME\_GET (“Get”) – return current simulator time

DCS\_CONTROL\_COMMAND\_DATETIME\_SET (“Set”) – set current simulator time

VARIANT \*pvData - pointer to variant for date and time information, type of variant: VT\_ARRAY | VT\_UI1. The variant stores byte buffer of the SYSTEMTIME time structure.

#### Error Return Code:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

##### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_DATETIME\_INVALID(0xD) – invalid date time format

#### VC++ Code Sample:

```
#include "DcsIO.h"
#include "BaileySim.h"
#include "BaileySim_i.c"
BOOL CopyVariantToBinary(BYTE*& pBuf,DWORD& dw, _variant_t& var)
{
    if( pBuf) return FALSE;
    if( var.vt == (VT_ARRAY | VT_UI1))
    {
        long lBound;
        SafeArrayGetLBound(var.parray, 1, &lBound);
        long uBound;
        SafeArrayGetUBound(var.parray, 1, &uBound);
        dw = uBound - lBound + 1;
        if(dw)
        {
            BYTE *p = NULL;
            p = new BYTE[dw];
            memset(p,0,dw);
            unsigned char FAR *pc = NULL;
            SafeArrayAccessData(var.parray, (void HUGEP* FAR*)&pc);
            _fmemcpy(p, pc, dw);
        }
    }
}
```

```

        SafeArrayUnaccessData(var.parray);
        pBuf = p;
        return TRUE;
    }
}
return FALSE;
}
BOOL GetTime()
{
    SYSTEMTIME time;
    HRESULT hr;
    _variant_t vError;
    _variant_t vData;
    ICDCSControl *pDCSControl = NULL;
    hr = CoCreateInstance(CLSID_CDcsControl, NULL, CLSCTX_SERVER,
IID_ICDCSControl, (LPVOID FAR *) &pDCSControl);
    if( hr == S_OK )
    {
        pDCSControl->DateTime ((_bstr_t)
DCS_CONTROL_COMMAND_DATETIME_GET, &vData, &vError);
        BYTE *p = NULL;
        DWORD dw = NULL;
        CopyVariantToBinary(p,dw, vData);
        memcpy(&time,p,sizeof(SYSTEMTIME));
        delete []p;
    }
    else return FALSE;
    return TRUE;
}

```

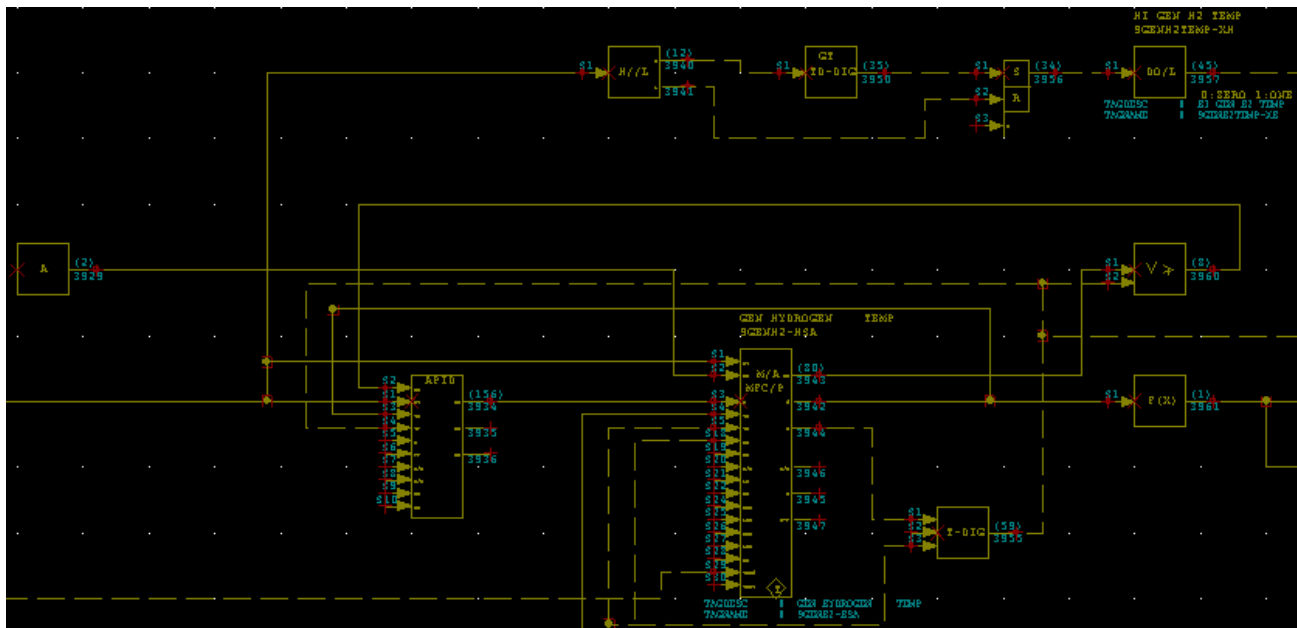
## 2.5 Process Data Read & Write Methods

The following two methods are provided:

- ReadBlockArray Method – This method supports:
  - Read array of block values, either digital or analog
  - Read Bailey Quality (Q) where available
  - Read array of arbitrary size
  - Read direct (from block address) and read indirect (from upstream block address)
  - Error checking to verify that expected and actual data type matches.
- WriteBlockArray Method – This method supports:
  - Write array of block values, either digital or analog
  - Write Bailey quality (Q) where applicable
  - Write array of arbitrary size
  - Written values are held until cleared or overwritten

### Typical Control Logic

Controller logic within the Bailey DCS controllers is provided in the form (primarily) of function blocks (e.g. lead/lag, station, analog input, digital output, PID, AND gate and so on). The following diagram illustrates some typical Bailey DCS control logic.



## Function Blocks

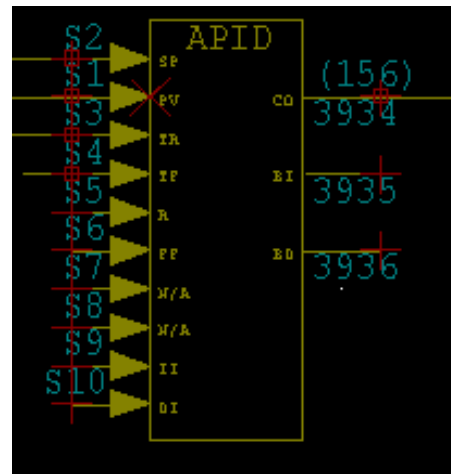
In total there are some 240+ function block types that have been used over the years within the Bailey DCS system. However, as some are primarily used in batch systems, or old and obsolete controllers, not all are supported within the Bailey DCS Simulator.

Refer to the Bailey DCS Users Manual for a list of all function codes supported by the Bailey DCS Simulator.

### A Typical Function Block

As illustrated here for an Advanced PID Block (Function Code FC156), each block is an object that has **Specifications, Inputs, Outputs**, and multiple **Block Addresses** associated with it.

**Specifications**, identified as S1, S2 and so on, may provide an address to an upstream **Input** (e.g. S1, S2, S3 and S4 in this diagram), or may provide a parameter for use inside the block (e.g. for FC156 specification S12, which is not



shown in the diagram, provides the proportional gain. For FC156 the table on the following page lists all available specifications.

The FC156 block also has three Outputs. In this case the block outputs are at Block Address 3934, 3935 and 3936. The following table lists all three Outputs. Note that the three outputs are all defined in terms of Base Block Address (N), which is usually referred to as the “block address” for an instance of the block, even though there are two other addresses. Note that each output is either Real or Boolean data type.

FC156 Outputs		
Blk	Type	Description
N	R	Control output with feed forward
N+1	B	Block increase flag: 0 = permit increase 1 = inhibit increase
N+2	B	Block decrease flag: 0 = permit decrease 1 = inhibit decrease

Not shown in the CAD drawings, or in the tables provided here, some (but not all) inputs and outputs also have a Quality state (called Q) associated with them in some operating conditions.

Specifications for FC156					
Spec	Tune	Default	Type	Range	Description
S1	N	5	I	Note 1	Block address of process variable
S2	N	5	I	Note 1	Block address of set point
S3	N	5	I	Note 1	Block address of track reference
S4	N	0	I	Note 1	Block address of track flag: 0 = track 1 = release
S5	N	5	I	Note 1	Block address of external or manual reset
S6	N	5	I	Note 1	Block address of feedforward signal
S7	N	5	I	Note 1	Spare real input
S8	N	0	I	Note 1	Spare boolean input
S9	N	0	I	Note 1	Block address of increase inhibit: 0 = normal 1 = prevent increase
S10	N	0	I	Note 1	Block address of decrease inhibit: 0 = normal 1 = prevent decrease
S11	Y	1.000	R	Full	Gain multiplier $K$
S12	Y	1.000	R	Full	Proportional gain $K_p$
S13	Y	0.000	R	Full	Integral reset resets/min. or manual reset time constant $K_I \times \text{min.}$
S14	Y	0.000	R	Full	Derivative rate action $K_D \times \text{min.}$
S15	Y	10.000	R	Full	Derivative lag constant $K_A$ (typically = 10)
S16	Y	105.000	R	Full	High output limit
S17	Y	-5.000	R	Full	Low output limit
S18	N	0	I	0,1,2 or 3	Algorithm type: 0 = classical 1 = noninteracting 2 = classical with external reset 3 = manual reset noninteracting
S19	N	0	I	0 or 1	Integral limit type: 0 = quick saturation recovery 1 = conventional saturation recovery
S20	Y	0	I	0 or 1	Set point modifier: 0 = normal 1 = integral only on set point change
S21	Y	0	I	0 or 1	Direction switch: 0 = reverse mode $\text{error} = SP - PV$ 1 = direct mode $\text{error} = PV - SP$
S22	Y	0.000	R	Full	Spare real parameter
S23	Y	0	I	Full	Spare integer parameter

## Process Host Writes to Block Address

The Bailey DCS API WriteBlockArray method provides the ability to write to any block output, to override the value that would otherwise be calculated within the block. In the case of analog and digital input blocks, this function simulates actual field input by setting the value at the selected block output address.

To write to any block address, the host requires:

- The address of the controller module within which the target block resides. This address is given as Loop (0 to 255), PCU (0 to 31), Module (0 to 31).
- The target Block address
- The data type Analog or Boolean
- The process value to write, and Quality (Q)

## Process Host Reads from Block

The Bailey DCS Simulator ReadBlockArray method supports both:

- Direct Read
- Indirect Read

### Direct Read

The ReadBlockArray method provides the ability to read any block output. To read the output value at any block address, the host requires:

- The L:P:M address of the controller module within which the target block resides.
- The target Block address
- The data type Analog or Boolean

### Indirect Read

The ReadBlockArray method also provides the ability to perform an indirect read of an upstream block output. To read the output value at any block address, the host requires:

- The L:P:M address of the controller module within which the target block resides.
- The target Block base address
- The specification number pointing to the upstream address
- The data type Analog or Boolean

Indirect Read is used to read from Analog and Digital field output blocks.

### Read Data and Quality (Q)

All read operations return the current process value (PV) of the address being read, whether directly or indirectly. In addition, Quality (Q) is returned also if available for the address being read.

## Input/Output Block Types

The following OUTPUT block types are supported, and the host would normally Read Indirect from these to get the analog or digital field output values:

- FC79 - Control Interface Slave
- FC83 - Digital Output Group
- FC149 - Analog Output/Slave
- Others

The following INPUT block types are supported, and the host would normally Write to these to simulate analog or digital field input values:

- FC84 - Digital Input Group
- FC132 - Analog Input/Slave
- FC216 - Enhanced Analog Input Definition



- FC242 - DSOE Digital Event Interface
- Others

Some block types, such as FC79 - Control Interface Slave, provides both field input and field output functions and so it would be normal for the host to both read from and write to this block.

### Program IO

The following INPUT block types are supported, and the host would normally Write to these to simulate program IO from embedded C or BASIC program segments within the Bailey DCS controller:

- FC93 - BASIC Real Output (Note 1)
- FC94 - BASIC Boolean Output (Note 1)
- FC137 - C and BASIC Program Real Output With Quality
- FC138 - C or BASIC Program Boolean Output With Quality

The most common usage for these program IO blocks is to provide an interface to foreign devices (e.g. Mark V turbine control, PLC, and others).

## Method: ReadBlockArray

### Syntax:

```
HRESULT ReadBlockArray(VARIANT vRequest, VARIANT
    *pvResponse, VARIANT *pvError)
```

### Where:

VARIANT vRequest – variant for requesting block addresses, type of variant: VT\_ARRAY | VT\_UI1. The variant stores byte buffer, which defines blocks addresses and operation for each block. The block data defined by the DcsIOReadBlock struct, described later.

VARIANT \*pvResponse – pointer to variant for returning values and errors, type of variant: VT\_ARRAY | VT\_UI1. The variant stores byte buffer, which includes blocks addresses, values and errors for each block. . The block data defined by the DcsIOBlockValue struct, described later.

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

### VC++ Code Sample:

```
#include "DcsIO.h"
#include "BaileySim.h"
#include "BaileySim_i.c"
BOOL CopyBinaryToVariant(BYTE *pBuf, DWORD dw, _variant_t& var)
{
    if( dw == 0 ) return FALSE;

    SAFEARRAYBOUND sabound[1];
    SAFEARRAY *psa;
```

```

var.Clear();
// Create safearray of unsigned char.
sabound[0].cElements = dw;
sabound[0].lLbound = 0;
psa = SafeArrayCreate(VT_UI1, 1, sabound);
if (psa)
{
    // Copy binary data into array
    unsigned char FAR *pc;
    SafeArrayAccessData(psa, (void HUGE_PTR FAR*)&pc);
    _fmemcpy(pc, pBuffer, dw);
    SafeArrayUnaccessData(psa);

    var.vt = VT_ARRAY | VT_UI1;
    var.parray = psa;
    return TRUE;
}
return FALSE;
}
BOOL CopyVariantToBinary(BYTE*& pBuffer,DWORD& dw, _variant_t& var)
{
    if (pBuffer) return FALSE;
    if (var.vt == (VT_ARRAY | VT_UI1))
    {
        long lBound;
        SafeArrayGetLBound(var.parray, 1, &lBound);
        long uBound;
        SafeArrayGetUBound(var.parray, 1, &uBound);
        dw = uBound - lBound + 1;
        if(dw)
        {
            BYTE *p = NULL;
            p = new BYTE[dw];
            memset(p,0,dw);
            unsigned char FAR *pc = NULL;
            SafeArrayAccessData(var.parray, (void HUGE_PTR FAR*)&pc);
            _fmemcpy(p, pc, dw);
            SafeArrayUnaccessData(var.parray);
            pBuffer = p;
            return TRUE;
        }
    }
    return FALSE;
}
BOOL ReadFunc()
{
    UINT iModuleCount = 1;
    UINT iBlockCount = 10;
    UINT iFormat = DCSIO_MESSAGE_FORMAT_READBLOCK;
    DWORD dwSize = 0;
    BYTE *pBuf = NULL;
    dwSize = sizeof(DcsIOMessage) +
        sizeof(DcsIOModule)*iModuleCount +
        sizeof(DcsIOReadBlock)*iBlockCount;
    pBuf = new BYTE[dwSize];
    memset(pBuf,0,dwSize);
    DcsIOMessage *pDcsIOMessage = (DcsIOMessage *)pBuf;
    pDcsIOMessage->m_iModuleCount = iModuleCount;
    pDcsIOMessage->m_iBlockCount = iBlockCount;
    pDcsIOMessage->m_iFormat = iFormat;
    pDcsIOMessage->m_iVersion = 1;
    // module
    DcsIOModule *pDcsIOModule = (DcsIOModule *) (pBuf +
        sizeof(DcsIOMessage));
}

```

```

pDcsIOModule->m_iLoop = 129;
pDcsIOModule->m_iPcu = 1;
pDcsIOModule->m_iModule = 3;
pDcsIOModule->m_iCount = iBlockCount;
pDcsIOModule->m_iOperation = DCSIO_MODULE_OPERATION_NONE;
// block
DcsIOReadBlock *pReadBlock = (DcsIOReadBlock *) (pBuf +
sizeof(DcsIOMessage) + sizeof(DcsIOModule));
UINT i = 0;
for( i = 0; i < iBlockCount; iBlockCount++)
{
    pReadBlock[i].m_iBlock = i+10;
    pReadBlock[i].m_iOperation =
DCSIO_READBLOCK_OPERATION_VALUE;
}
_variant_t vRequest;
_variant_t vResponse;
_variant_t vError;
CopyBinaryToVariant(pBuf,dwSize, vRequest);
// call debugger
delete []pBuf;
pBuf = NULL;
dwSize = 0;
HRESULT hr;
vResponse.Clear();
ICDcsControl *pDcsControl = NULL;
hr = CoCreateInstance(CLSID_CDcsControl, NULL, CLSCTX_SERVER,
IID_ICDcsControl, (LPVOID FAR *) &pDcsControl);
if( hr == S_OK )
{
    hr = pDcsControl->ReadBlockArray(vRequest,&vResponse, &vError);
    if( hr == S_OK )
    {
        CopyVariantToBinary(pBuf,dwSize, vResponse);
        pDcsIOMessage = (DcsIOMessage *)pBuf;
        iModuleCount = pDcsIOMessage->m_iModuleCount;
        iBlockCount = pDcsIOMessage->m_iBlockCount;
        iFormat = pDcsIOMessage->m_iFormat;
        pDcsIOModule = (DcsIOModule *) (pBuf + sizeof(DcsIOMessage));
        UINT iLoop = pDcsIOModule->m_iLoop;
        UINT iPcu = pDcsIOModule->m_iPcu;
        UINT iModule = pDcsIOModule->m_iModule;
        iBlockCount = pDcsIOModule->m_iCount;
        // block
        DcsIOBlockValue *pValue = (DcsIOBlockValue *) (pBuf +
sizeof(DcsIOMessage) + sizeof(DcsIOModule));
        UINT i = 0;
        for( i = 0; i < iBlockCount; iBlockCount++)
        {
            int iBlock = pValue[i].m_iBlock;
            UINT iOperation = pValue[i].m_iOperation;
            UINT iError = pValue[i].m_error;
            double dValue = (double)pValue[i].m_fValue;
            UINT iQuality = pValue[i].m_quality;
        }
        delete []pBuf;
    }
    pDcsControl->Release();
}
else return FALSE;
vResponse.Clear();
vError.Clear();
return TRUE;
}

```

## Method: WriteBlockArray

### Syntax:

```
HRESULT WriteBlockArray(VARIANT vRequest,VARIANT
*pvResponse,VARIANT *pvError);
```

### Where:

VARIANT vRequest – variant for block addresses and written values, type of variant: VT\_ARRAY | VT\_UI1. The variant stores byte buffer defining block addresses, operations and block values. The block data, defined by the DcsIOBlockValue struct, is described later.

VARIANT \*pvResponse – pointer to variant for returning errors, type of variant: VT\_ARRAY | VT\_UI1. The variant stores byte buffer, which includes block addresses and errors for each block. . The block data, defined by the DcsIOBlockError struct, is described later. Method WriteBlockArray returns block data only for errors.

### Error Return Codes

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

## Data Structures

### Message Format for ReadBlockArray and WriteBlockArray.

Each message stored in vRequest and vResponse variants for ReadBlockArray and Write block array has similar format:

```
<message header><module data><module data>...<module data>
```

Message header is defined as a C/C++ structure( all structures and constants are defined in the header file DcsIO.h)

```
typedef struct DCSIO_MESSAGE_TAG
{
    USHORT m_iVersion;
    USHORT m_iFormat;
    USHORT m_iModuleCount;
    USHORT m_iBlockCount;
    USHORT m_iOperation;
} DcsIOMessage;
```

Where:

m\_iVersion – message version, now is 1

m\_iFormat – format of the block data,

DCSIO\_MESSAGE\_FORMAT\_READBLOCK(1) – request to read blocks, the block data defined by the DcsIOReadBlock struct

DCSIO\_MESSAGE\_FORMAT\_BLOCKVALUE(2) – response with block values or request to write blocks, the block data defined by the DcsIOBlockValue struct

DCSIO\_MESSAGE\_FORMAT\_BLOCKERROR(3) – returned block’s errors, the block data defined by the DcsIOBlockError struct

m\_iModuleCount – number of the modules in the message

m\_iBlockCount – number of blocks in all modules

m\_iOperation – type of the operation for all modules in the message

DCSIO\_MESSAGE\_OPERATION\_NONE (0) - no operation

DCSIO\_ENABLE\_ALL\_ALARMS (1) - Enable All Alarms in All Modules

DCSIO\_DISABLE\_ALL\_ALARMS (2) - Disable All Alarms in All Modules. Will not clear alarms already at console.

DCSIO\_ACK\_ALL\_ALARMS (4) - Acknowledge All Alarms on all CIU that are online

### Example to enable all alarms

```

CDcsIOMsg<DcsIOBlockValue>
request(0,0,DCSIO_MESSAGE_FORMAT_BLOCKVALUE,DCSIO_ENAB
LE_ALL_ALARMS);
BYTE *pResponse = NULL;
DWORD dwResponse = NULL;
HRESULT hr;
_variant_t vRequest;
_variant_t vResponse;
_variant_t vError;
ICDcsControl *pDcsControl = NULL;
hr = CoCreateInstance(CLSID_CDcsControl, NULL, CLSCTX_SERVER,
IID_ICDcsControl, (LPVOID FAR *) &pDcsControl);
if( hr == S_OK )
{
    hr = pDcsControl->WriteBlockArray(vRequest,&vResponse,
&vError);
    if( hr == S_OK )
    {
        if( hr == S_OK )
        {
            CopyVariantToBinary(pResponse,dwResponse,
vResponse);
        }
    }
    vResponse.Clear();
    vError.Clear();
    pDcsControl->Release();
}

```

### Data Structures to Define Module

Each <module data> buffer has following format:

<module header><block data><block data>...<block data>

Module header is defined as:

```

typedef struct DCSIO_MODULE_TAG
{
    USHORT m_iLoop;
    USHORT m_iPcu;
}

```

```

USHORT m_iModule;
USHORT m_iCount;
USHORT m_iOperation;
USHORT m_iError;
} DcsIOModule;

```

Where:

m\_iLoop – loop address  
m\_iPcu – pcu address  
m\_iModule – module address  
m\_iCount – number of blocks in module data buffer  
m\_iOperation – operation for this module  
DCSIO\_MODULE\_OPERATION\_NONE(0) – no operation  
DCSIO\_MODULE\_OPERATION\_CLEAR(1) – unforce all forced values in the module, currently not supported  
m\_iError – module error  
DCSIO\_MODULE\_ERROR\_NONE(0) – no error  
DCSIO\_MODULE\_ERROR\_INVALIDADDRESS(1) – invalid module address

### Data Structures for Requesting Block Value:

```

typedef struct DCSIO_READBLOCK_TAG
{
    unsigned short m_iBlock;
    unsigned char m_iOperation;
    unsigned char m_iNum;
} DcsIOReadBlock;

```

Where:

m\_iBlock – block address  
m\_iOperation – block operation  
DCSIO\_READBLOCK\_OPERATION\_VALUE(0) – read block output, if block output is forced, read forced block output  
DCSIO\_READBLOCK\_OPERATION\_FORCED(1) – read forced block output, if block output is not forced, returns error  
DCSIO\_READBLOCK\_OPERATION\_NORMAL( 2) – read block output produced by function code, even block output is forced  
m\_iNum – reserved for indirect read

### Data Structures for returned or written block value:

```

typedef struct DCSIO_BLOCKVALUE_TAG
{
    unsigned short m_iBlock;
    unsigned short m_iOperation;
    unsigned m_quality:2;
    unsigned m_spare:6;
}

```

```

    unsigned short m_error;
    float m_fValue;
} DcsIOBlockValue;

```

Where:

m\_iBlock – block address

m\_iOperation – block operation

DCSIO\_BLOCKVALUE\_OPERATION\_WRITE\_FIELDIO(1) – write Field IO block output

DCSIO\_BLOCKVALUE\_OPERATION\_WRITE\_PROGRAMIO(2) – program io

DCSIO\_BLOCKVALUE\_OPERATION\_WRITE\_FORCE(3) – force block output

DCSIO\_BLOCKVALUE\_OPERATION\_WRITE\_CLEAR(4) – if forced, cancel forcing block output

m\_quality:2 – quality

DCSIO\_BLOCKVALUE\_QUALITY\_GOOD(0) – good quality

DCSIO\_BLOCKVALUE\_QUALITY\_BAD(1) – bad quality

DCSIO\_BLOCKVALUE\_QUALITY\_UNCHANGED(2) – do not change quality

m\_spare:6 – not used

m\_error – block error

DCSIO\_BLOCK\_ERROR\_NONE(0) – no error

DCSIO\_BLOCK\_ERROR\_INVALIDADDRESS(0x1) – invalid block address

DCSIO\_BLOCK\_ERROR\_INVALIDFC(0x2) – invalid function code type

DCSIO\_BLOCK\_ERROR\_INVALIDVALUE(0x4) – invalid value

DCSIO\_BLOCK\_ERROR\_INVALIDQUALITY(0x8) – invalid quality

DCSIO\_BLOCK\_ERROR\_FIELDIONOTSUPPORTED(0x10) – field io is not supported by this function code

DCSIO\_BLOCK\_ERROR\_INVALIDOPERATION(0x20) – invalid block operation

DCSIO\_BLOCK\_ERROR\_INVALIDSPEC (0x40) - invalid spec number

DCSIO\_BLOCK\_WARNING\_NOTDIGITAL (0x80) - value is not 0 or 1

DCSIO\_BLOCK\_WARNING\_OUTOFRANGE (0x03) - value is out of range (i.e. not between expected ZERO and SPAN)

DCSIO\_BLOCK\_WARNING\_NOTEXISTS (0x05) - block is not defined for this address

m\_fValue – block value

### Data Structures for returned block error:

```

typedef struct DCSIO_BLOCKERROR_TAG
{
    unsigned short m_iBlock;

```

```
        unsigned short m_iError;  
    } DcsIOBlockError;
```

Where:

m\_iBlock – block address

m\_iError – block error, see error for DcsIOBlockValue struct.



---

## 2.6 DCS State File Functions

The state file functions support save and restore of DCS process state (sometimes called IC – Initial Condition), as well as some related housekeeping functions.

### Overview

The DcsState method supports the following specific operations:

- **Save** – Pause execution (if required) and save current process state, with filename provided at method call. Remain in pause state. Generates error message if state file by same name already exists.
- **Save/Continue** – Pause execution (if required) and save current process state, with filename provided at method call. Continue execution. Generates error message if state file by same name already exists.
- **Overwrite** – Pause execution (if required) and save current process state, with filename provided at method call. Remain in pause state. Overwrites (without error message) existing state file by same name if file already exists.
- **Overwrite/Continue** – Pause execution (if required) and save current process state, with filename provided at method call. Continue execution. Overwrites (without error message) existing state file by same name if file already exists.
- **Restore** – Pause execution (if required) and restore a state file, with filename provided at method call. Remain in pause state.
- **Restore/Continue** – Pause execution (if required) and restore a state file, with filename provided at method call. Continue execution.
- **List** – returns a list of the filenames of all saved state files
- **Delete** – delete one specific state file, with filename provided at method call.

### What is Saved to, and Restored from, State File

The following are saved to state file and restored upon DCS file restore:

- Internal state of all blocks
- Last Analog and Digital input values written via API
- Last Analog and Digital output values

## The DCS process state file

The Bailey DCS Simulator supports an unlimited number of DCS state files, constrained only by the disk space available. The DCS state file takes file extension *.DCS*. Typical DCS state file size is 300 to 400 bytes per FC block.

## Path for DCS State Files

The path at which all DCS process state files will be saved is defined at the *Default.INI* file. If no path is defined, all DCS state files will be stored at the *.. \Bailey DCS Simulator\* directory, which is not recommended.

## Method: DcsState

### Syntax:

```
HRESULT DcsState(BSTR szOperation, VARIANT *pvData, VARIANT *pvError)
```

### Where:

BSTR szOperation – operation. Values:

DCS\_CONTROL\_STATE\_FILE\_SAVE (“Save”) – pause (if required) and save current state without continue

DCS\_CONTROL\_STATE\_FILE\_SAVE\_CONTINUE (“Save/Continue”) – pause (if required) and save current state and continue execution

DCS\_CONTROL\_STATE\_FILE\_OVERWRITE (“Overwrite”) – pause (if required) and save current state (with overwrite) without continue

DCS\_CONTROL\_STATE\_FILE\_OVERWRITE\_CONTINUE (“Overwrite/Continue”) - pause (if required) and save current state (with overwrite) and continue execution

DCS\_CONTROL\_STATE\_FILE\_RESTORE (“Restore”) – pause (if required) and restore state from file without continue

DCS\_CONTROL\_STATE\_FILE\_RESTORE\_CONTINUE (“Restore/Continue”) – pause (if required) and restore state from file and continue execution

DCS\_CONTROL\_STATE\_FILE\_LIST (“List”) – returns list of the saved state file’s names

DCS\_CONTROL\_STATE\_FILE\_DELETE (“Delete”) – delete a state file

VARIANT \*pvData - pointer to variant, type of variant: VT\_BSTR for “Save”, “Overwrite”, “Restore” and “Delete” operations, stores file name.

By default, the state file is restored, without restoring specification values of function codes. Specification values stay as they were loaded during start up of the Bailey DCS Simulator from the CFG module files. However, all specification values are saved in the state file and can be restored if required.

---

**NOTE:** If specifications are restored from the state file, they may over-write valid specifications that have been set from the operator console or EWS. This function should be used with caution, and with awareness of the implications of its use.

---

To restore a state file, complete with all saved specifications, the name of the state file set in the VARIANT \*pvData must be appended with character string “/Specs” separated by the space.

For example:

“*FileName.dcs*” has to be changed to “*FileName.dcs /Specs*”

VT\_ARRAY | VT\_VARIANT for “List” operation, stores array of variants for file names.

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error

DCS\_CONTROL\_STATEFILE\_EXISTS(0x7) – a state file exists

DCS\_CONTROL\_STATEFILE\_NOT\_EXISTS(0x8) – a state file does not exist

DCS\_CONTROL\_STATEFILE\_CANNOT\_DELETE(0xC) – state file can not be deleted

### VC++ Code Sample:

```
#include "DcsIO.h"
#include "BaileySim.h"
#include "BaileySim_i.c"
BOOL GetStateFileList()
{
    _variant_t vError;
    _bstr_t szFunc(DCS_CONTROL_STATE_FILE_LIST);
    _variant_t vData;
    vData.Clear();
    ICDcsControl *pDcsControl = NULL;
    HRESULT hr = CoCreateInstance(CLSID_CDcsControl, NULL,
    CLSCTX_SERVER, IID_ICDcsControl, (LPVOID FAR *) &pDcsControl);
    if( hr == S_OK )
    {
        hr = pDcsControl->DcsState(szFunc,&vData,&vError);
        long x = 0;          // Get property from array
        _variant_t vValue;
        while(S_OK == SafeArrayGetElement(vData.parray, &x, &vValue))
        {
            _bstr_t sz = vValue.bstrVal;
            char filename[128];
            strcpy(filename,sz);
            vValue.Clear();
            x++;
        } // End while()
    }
    else return FALSE;
    return TRUE;
}
```

### Performance Notes

To improve performance at large DCS file size, the DCS process state is buffered in RAM at DcsState(Save) method call so that the method call can return quickly. Once the call has returned, the buffered DCS state file data is written to file.

In system with 95 controller CFG files and ~170,000 blocks, with a DCS state file size of ~50MB, it takes about 250 milliseconds to return from DcsState(Save) and DcsState(Save/Continue) methods. In smaller systems the return will be faster.

## 2.7 Command Log and Replay Functions

The command log functions support the time tagged logging of operator commands by an external host computer so that these can be “played back” under host control at a later time.

To use this function the normal sequence of events would be:

- RESET the command buffer. This clears any previously saved commands from the buffer. The command buffer is also cleared at startup of Bailey DCS Simulator.
- START the command log. This causes incoming operator commands, to any emulated CIU, to be logged to a command buffer.
- GET array of command log buffer contents at any time for storage external to the Bailey DCS Simulator. Commands that are retrieved via the GET operation are cleared from the command buffer at that time. Buffer will continue to fill after this function. Each command record will contain a HEX structure for replay.
- STOP the command log will stop the logging of additional commands. It is advisable to GET any residual commands in the command buffer at this time and then RESET the command buffer to be ready for the next time logging is required.
- REPLAY logged commands one at a time when ready.

Please note that this command logging function is designed to be under external control. This means that the REPLAY function requires a logged command to have been first retrieved from the command buffer by the external application, stored externally, and then replayed at the right time. The Bailey DCS Simulator acts as a slave in this regard, and simply “plays” the operator command at the instant the external host commands it to.

### Command Buffer Details

The maximum number of commands that can be stored in the command buffer is 1,024. These commands are stored in a circular buffer such that the oldest commands are overwritten by the newest commands if the buffer capacity is exceeded. As a result, if a maximum “operator command rate of 5 operator commands per second is assumed, then the buffer will fill in 204 seconds. As a result, it is recommended that the CommandLog(Get) operation be performed at 60 second intervals during Command Logging to ensure that this buffer never overflows.

## Method: CommandLog operations

### Syntax:

```
HRESULT CommandLog(BSTR szOperation,VARIANT *pvData,VARIANT *pvError);
```

### Where:

BSTR szOperation – operation. Values:

DCS\_CONTROL\_COMMAND\_LOG\_START (“Start”) – starts command log

DCS\_CONTROL\_COMMAND\_LOG\_STOP (“Stop”) – stops command log

DCS\_CONTROL\_COMMAND\_LOG\_GET (“Get”) – retrieves current command log

DCS\_CONTROL\_COMMAND\_LOG\_RESET (“Reset”) – resets command log

DCS\_CONTROL\_COMMAND\_LOG\_REPLAY (“Replay”) – replays command log

VARIANT \*pvData - pointer to variant, type of variant: VT\_ARRAY | VT\_UI1 for “Get” and “Replay” operations. Variant stores byte array for command log. Byte array is defined as a CommandLogMessage struct, see DCSIO.H file, defined as:

```
typedef struct COMMAND_LOG_MESSAGE_TAG
{
    USHORT m_iVersion;
    USHORT m_iFormat;
    USHORT m_iCount;
    CommandLogRecord m_record[1];
} CommandLogMessage;
```

Where:

m\_iVersion – version of the message, now 1

m\_iFormat – format of the CommandLogRecord struct, now 1

m\_iCount – number of logged commands

CommandLogRecord m\_record[] – array of the logged commands, currently maximum size is 1024

CommandLogRecord is defined as struct:

```
#define COMMAND_LOG_RECORD_BUF_LEN 8
typedef struct COMMAND_LOG_RECORD_TAG
{
    SYSTEMTIME m_time;
    USHORT m_usLoop;
    USHORT m_usPcu;
    USHORT m_usModule;
    USHORT m_usBlock;
    int m_iMsgType;
    BYTE m_buf[COMMAND_LOG_RECORD_BUF_LEN];
} CommandLogRecord;
```

Where:

m\_time – time, when the command was executed  
 m\_usLoop – loop address from 0 to 255  
 m\_usPcu – pcu address from 0 to 31  
 m\_usModule – module address from 0 to 31  
 m\_usBlock – block address from 1 to 9999  
 m\_iMsgType – message type  
 m\_buf – byte buffer, includes content of executed command

### Error Return Codes:

VARIANT \*pvError – pointer to variant for error, type of variant VT\_I2

#### Return code:

DCS\_CONTROL\_NO\_ERROR(0) – no error  
 DCS\_CONTROL\_COMMANDLOG\_STARTED(0x9) – command log already started  
 DCS\_CONTROL\_COMMANDLOG\_STOPPED(0xA) – command log already stopped  
 DCS\_CONTROL\_COMMANDLOG\_INVALIDADDRESS (0xE) – invalid block address  
 DCS\_CONTROL\_COMMANDLOG\_INVALIDFC (0xF) – invalid function code type  
 DCS\_CONTROL\_COMMANDLOG\_INVALIDVALUE (0x10) – invalid value

### VC++ Code Sample:

```
#include "DcsIO.h"
#include "BaileySim.h"
#include "BaileySim_i.c"
BYTE *pLog;
DWORD dwLog;
BOOL GetCommandLog()
{
    pLog = NULL;
    dwLog = 0;
    _variant_t vError;
    _variant_t vData;
    _bstr_t szOperation(DCS_CONTROL_COMMAND_LOG_GET);
    vData.Clear();
    ICDCSControl *pDCSControl = NULL;
    HRESULT hr = CoCreateInstance(CLSID_CDcsControl, NULL,
    CLSCTX_SERVER, IID_ICDCSControl, (LPVOID FAR *) &pDCSControl);
    if (hr == S_OK)
    {
        hr = pDCSControl->CommandLog(szOperation, &vData, &vError);
        CopyVariantToBinary(pLog, dwLog, vData);
        CommandLogMessage *pCommandLog = (CommandLogMessage *)pLog;
        UINT iCount = pCommandLog->m_iCount;
        pDCSControl->Release();
    }
    return TRUE;
}
BOOL Replay(UINT iNum)
{
    if (pLog)
    {
        _variant_t vError;
        _variant_t vData;

        _bstr_t szOperation(DCS_CONTROL_COMMAND_LOG_REPLAY);
```

```

vData.Clear();
CommandLogMessage log;
CommandLogMessage *p = (CommandLogMessage *)pLog;
if( iNum < p->m_iCount )
{
    log.m_iCount = 1;
    memcpy(&log.m_record[0],&p-
>m_record[iNum],sizeof(CommandLogRecord));
    CopyBinaryToVariant((BYTE *)&log,sizeof(CommandLogMessage), vData);
    //CopyBinaryToVariant(pLog,dwLog, vData);
    ICDcsControl *pDcsControl = NULL;
    HRESULT hr = CoCreateInstance(CLSID_CDcsControl, NULL,
    CLSCTX_SERVER, IID_ICDcsControl, (LPVOID FAR *) &pDcsControl);
    if( hr == S_OK )
    {
        hr = pDcsControl->CommandLog(szOperation, &vData, &vError);
        pDcsControl->Release();
    }
}
return TRUE;
}
return FALSE;
}

```

## 3 Usage Scenarios

This section describes typical examples of how the simulator host would interact with the Bailey DCS Simulator during operation.

### 3.1 Simulator Startup

Method(Operation)	Notes
DateTime(Set)	Synchronize time
Pause	Pause operations
DCSState(Restore)	Restore required state file
Continue	Continue operations
WriteBlockArray	Write Field IO array (from Host to DCS)
ReadBlockArray	Read Field IO Array (From DCS to Host)
<b>Alternative (Pause and Continue automatic via DCSState(Restore))</b>	
DateTime(Set)	Synchronize time
DCSState(Restore/Continue)	Restore required state file and continue
WriteBlockArray	Write Field IO array (from Host to DCS)
ReadBlockArray	Read Field IO Array (From DCS to Host)

### 3.2 Simulator Shutdown

Method(Operation)	Notes
Pause	Stop simulator execution
DCSState(Save)	Save state (if required) and pause
CommandLog(Stop)	Stop command log (if required)
CommandLog(Get)	Retrieve buffered commands (if required)
	STOP Bailey DCS Simulator service

### 3.3 Normal Quiescent Simulator Operation

Method(Operation)	Notes
WriteBlockArray	Write Field IO array (from Host to DCS)
ReadBlockArray	Read Field IO Array (From DCS to Host)
WriteBlockArray	Write Field IO array (from Host to DCS)
ReadBlockArray	Read Field IO Array (From DCS to Host)



### 3.4 Command Logging

Method(Operation)	Notes
<b>Logging</b>	
CommandLog(Reset)	Clear any old command log contents
CommandLog(Start)	Start command log
CommandLog(Get)	Retrieve and file command log buffer entries
CommandLog(Get)	
CommandLog(Get)	
CommandLog(Get)	
CommandLog(Get)	
CommandLog(Stop)	Stop Command Log
CommandLog(Get)	Retrieve and file command log buffer entries
<b>Replay</b>	
DCSState(Restore/Continue)	Restore required state file and continue
CommandLog(Replay)	Replay commands in sequence at correct time after Continue.
CommandLog(Replay)	
CommandLog(Replay)	

### 3.5 Backtrack

Method(Operation)	Notes
DCSState(Save/continue)	Save State at time T and continue
DCSState(Save/continue)	Save State at time T+1 and continue
DCSState(Save/continue)	Save State at time T+2 and continue
<b>Backtrack</b>	
DCSState(Restore/Continue)	Restore desired state file and continue
WriteBlockArray	Write Field IO array (from Host to DCS)
ReadBlockArray	Read Field IO Array (From DCS to Host)