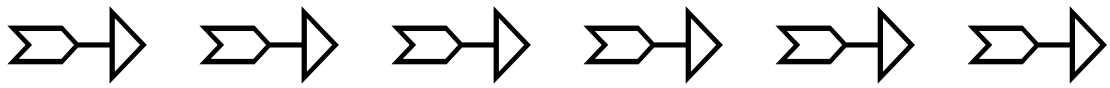
DAI

DARTS Application Interface

User Manual

Version 1.00



Document Number E-UM-012-A0

September 1992

Contents **Page**

Introduction:

The DAI is a set of functions which implement the Datataker DARTS protocol for a host application. The functions provide an easy and flexible interface that enables an application to communicate with the Datataker range of data loggers using the DARTS protocol.

When using these functions the application doesn't need any knowledge of how the DARTS protocol works, this is all handled by the DAI functions. Using the DAI, and consequently the DARTS protocol, enables the application to implement an "error free" communications link to the Datataker. This is very useful for modem links, radio modem links, noisy local links (for example long cable length), or where information is critical and has to be guaranteed. On the down side there is a loss in data throughput as extra protocol information has to travel in both directions on the link to enable the protocol to work. The loss in through put would normally be around 30 percent.

The DAI provides easy to use functions to open and close asynch communications port, initialise a DARTS session, set the Datataker into or out of DARTS protocol mode, send commands and receive data.

The DAI is a set of C functions that are provided as a library which can be linked in with your application.(See "Compilation Notes", Page 5, for more details)

Files provided:

There are four versions of the DAI library for different memory models. There is also a header file for using the DAI functions. The complete source code for the darts functions and the interface to the Blaise asynch manager are also provided.

Two example applications are provided. One is a data down load utility, the other is a simple terminal. The source code for both applications is provided. Both applications use the DAI functions to provide error free conversations with the attached Datataker and are provided as example applications.

Summary of files provided

DAI_MCS.LIB	DAI Library for Microsoft C - small model
DAI_MCC.LIB	DAI Library for Microsoft C - compact model
DAI_MCM.LIB	DAI Library for Microsoft C - medium model
DAI_MCL.LIB	DAI Library for Microsoft C - large model
DAI.H	Include file for using DAI functions
ASY_INTF.H	Include file for using asynch interface functions
ASY_INTF.C	Source for asynch manager interface functions
DAI.C	Source for DAI functions
DAI_UTIL.H	Include file with common definitions
DOWNLOAD.EXE	Download utility program
TERMINAL.EXE	Terminal utility program
DOWNLOAD.C	Source code for download utility
TERMINAL.C	Source code for terminal utility

Installation:

There are no special installation requirements. All the files on the diskette supplied are in standard DOS format. They can be copied directly to your hard disk. Thus no installation program is provided, simply copy the required files using the DOS "COPY" command.

Compilation notes:

This version of the DAI was written for a PC host. It was compiled with Microsoft C Version 6.00 and the Blaise Asynch Manager V3.00. You will require a copy of Blaise Asynch Manager to be able to link your application. It is possible to provide your own or other asynch manager - see "Asynch manager interface" - Page 22. The DAI functions only use standard ANSI library functions and so should be portable to other ANSI conforming compilers. To do this you must re-compile the source code provided to create a library compatible with your environment.

You should be able to obtain the Blaise Asynch Manager from your local programming utilities software distributer or you can contact Blaise Computing Inc. directly to locate your nearest distributor.

Blaise Computing Inc.
819 Bancroft Way
Berkeley, CA 94710-2226

Phone: 510-540-5441
Fax: 510-540-1938.

The DAI is provided in a library format. There are four different libraries, only one of which is used depending on the memory model the application is using.

The four libraries are

DAI_MCS.LIB	Microsoft C - small model
DAI_MCC.LIB	Microsoft C - compact model
DAI_MCM.LIB	Microsoft C - medium model
DAI_MCL.LIB	Microsoft C - large model

To link an application the appropriate DAI library must be specified and also the appropriate asynch manager library and the special asynch manager object file ASYNCH.OBJ.

The appropriate Blaise asynch manager libraries are

ASY_MCS.LIB	Microsoft C - small model
ASY_MCC.LIB	Microsoft C - compact model
ASY_MCM.LIB	Microsoft C - medium model
ASY_MCL.LIB	Microsoft C - large model

An example link command for a small model program in the Microsoft C environment would be;

```
LINK APPLICATION+ASYNCH,,,ASY_MCS+DAI_MCS;
```

Usage and general guidelines:

The DAI functions use a structure known as a Session to help manage a connection to a Datataker. The DAI functions are not limited to just one session (connection), many sessions can be handled at once. This allows an application to communicate with more than one Datataker simultaneously.

The limitation to the number of simultaneous sessions is the number of serial communications ports available. In the standard version of the DAI functions up to four communications ports are supported (COM1, COM2, COM3 and COM4) and so up to four sessions can be run together.

The session management structure contains all of the information relating to a session to enable the DAI functions to process the characters being sent and received. The session structure contains information such as the communications port number, the baud rate to use, the data received and transmitted and various internal states and flags. This session structure is passed by reference (ie. a pointer to the session structure) to all DAI functions so that they can keep track of and inform other functions of various events and conditions relating to the session.

The session structure must be defined by the application. This can be done by defining a structure on the stack (local to a function - automatic) or data area memory (local to module or global data - static or global). The structure could alternately be allocated from dynamic memory (using malloc or similar).

A communications port needs to be opened before starting. This should be done with **DAIOpenPort**. This will set the chosen baud rate and fix the other line settings suitable for connection to a Datataker. It will also set up the interrupts for handling sending and receiving characters on the communications port.

After the session structure is defined/allocated, and the communications port has been opened the session structure must be initialised using the function **DAIInit**.

If the Datataker is connected remotely then it may be necessary to establish that connection. For example a dial-up modem link. If this is necessary then the next step is to establish that connection. For this purpose the **DAISendDirect** and **DAIReadDirect** functions enable you to send and receive characters directly to and from the communications port, by-passing the DARTS protocol. In a dial-up modem link the modem dial commands can be sent directly and the result code can be monitored to determine when the connection has been established or if the connection has failed. See "Modem Connections"- Page 22 for more details.

The next step is to turn on the DARTS protocol in the attached Datataker. This can be done simply by calling the function **DAISetMode**. Once the Datataker is in DARTS protocol mode then the application is free to send commands and receive data using the send and receive functions.

To send commands to the Datataker there are two methods. Firstly the **DAISendCmdWait** function can be used. This is the simplest method. This function will send and wait for the command to be accepted by the Datataker before returning. During this time no data will be received. For applications where no data will be being received while sending commands then this method is best. However if the application needs to process something else while waiting for the command to be accepted then the DAISendCmdWait function is not appropriate as it sits and waits without returning control, this could be as long as 30 seconds or more.

The second method is to set up the command to send using the **DAISendCmd** function. And then poll for the result of the send operation using the **DAICmdStatus** function. With this method the **DAICmdStatus** function must be called often to enable the DARTS session to be processed in a timely fashion. If the function is not called often enough the Datataker may time-out on message sends and fail to return data. Between polling for the transmission status any other jobs can be carried out. The time-out for a message retry is about 5 seconds, so any other processing should be kept under this figure to avoid re transmissions. The **DAICmdIdle** function can be called after reading the final status of message transmission, either DAISentOK or DAITooManyRetries, so that it can then tell that the status now no longer applies to any message outstanding.

To receive data the application should call the **DAIReadData** function frequently. This function does two jobs. Firstly it will return any received data from the session. Secondly it will do any actions necessary to process the DARTS session. This second action is the same as the **DAICmdStatus** function , either or both of which must be called frequently.

Once the application has finished with the session it need only close the port by calling **DAIClosePort**. If necessary the Datataker can be set out of DARTS protocol mode when finished, or at any other time, by calling **DAISetMode**.

One other utility function has not been described so far. The function **DAINewBaudRate** must be called if the baud rate is changed during the course of a session. This is not very likely to be required but is provided just in case.

DAI Facilities summary:

The following types, macros, enumerated types are provided for use with the DAI functions;

Macros:

`DAI_MAX_DATA_LEN`	the maximum message length that can be sent/received

Structures:

`DAISession`	the DARTS session management structure
`DAISessionPtr`	a pointer to a DARTS session management structure
`DAI_SESSION_SIZE`	the size (in bytes) of the session management structure

Enumerated types:

`DAITxCmdState`	status of message being sent. Returned by **DAICmdStatus** function
`DAITxing`	message still being sent
`DAISentOK`	message sent ok
`DAITooManyRetries`	couldn't send message - too many retries
`DAITransmitIdle`	transmitter idle
`DAIProtMode`	turn on/off protocol mode on Datataker, passed to **DAISetMode**
`DAIProtocolOff`	turn off protocol mode
`DAIProtocolOn`	turn on protocol mode

Functions:

`DAIOpenPort`	Open a comms port at a baud rate for use with a Datataker
`DAIClosePort`	Close a comms port
`DAIInit`	Initialised session management structure
`DAISetMode`	Sets/resets DARTS protocol mode on a Datataker
`DAIReadData`	Reads any waiting messages sent via DARTS protocol
`DAISendCmdWait`	Send command to Datataker using DARTS protocol
`DAISendCmd`	Sets up a command to send using DARTS protocol
`DAICmdStatus`	Returns status of command being sent via DARTS
`DAICmdIdle`	Sets command sent status to idle
`DAINewBaudRate`	Changes session baud rate setting during a session
`DAISendDirect`	Sends a string to comm port directly, by-passing DARTS protocol
`DAIReadDirect`	Reads a single character, if available, directly from a comm port

DAI Function Reference

The following section describes each of the provided DAI functions. Each function is listed on a separate page. Example code fragments are given for all functions.

DAIClosePort

Description Close a communications port after use with Datataker

```
#include <dai.h>

int DAIClosePort(int CommPort);
```

CommPort communications port number to close (1,2,3,4)

Return Value TRUE (1) is returned if the port is successfully closed. If unsuccessful, because the port was not open or the port number was invalid then FALSE (0) is returned.

Remarks This function will close a previously opened communications port. Any port that has been opened should be closed before exiting the application or else the interrupt drivers will be undefined and will cause problems. The port must have already been opened using the **DAIOpenPort** function before it can be closed. The only supported communications port numbers are 1 to 4 (COM1 to COM4 in PC terminology)

See Also **DAIOpenPort**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
/* attempt to close communications port */
if (!DAIClosePort(1))
   {
   printf("\n<Could not close COM1>\n");
   }
.
.
```

DAICmdIdle

Description Sets the sent command status to DAITransmitIdle

```
#include <dai.h>

void DAICmdIdle(DAISessionPtr Session);
```

Session pointer to session to set status on

Return Value None.

Remarks Once the system has read the completion status of the last command as sent then it can call this function to reset the last command status back to DAITransmitIdle.

This function doesn't have to be used. It can be useful when managing commands sent and outstanding, in an application.

See Also **DAISendCmd, DAICmdStatus**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
DAITxCmdState CmdStatus;
.
.
/* check on status of any command being sent */
CmdStatus = DAICmdStatus(Session);

/* if not idle and not currently transmitting then must
   have just finished sending a command */
if ((CmdStatus != DAITransmitIdle) &&
    (CmdStatus != DAITxing))
   {
   /* check if failed to send command */
   if (CmdStatus == DAITooManyRetries)
      {
      printf("\n<Unable to send command>\n");
      }

   /* set command status to idle */
   DAICmdIdle(Session);
   }
.
.
```

DAICmdStatus

Description Returns the command being/just sent current status

```
#include <dai.h>

DAITxCmdState DAICmdStatus(DAISessionPtr Session);
```

Session pointer to session to return status for

Return Value This function returns an enumerated type DAITxCmdState. Which can take on any of the following values

DAITxing	still transmitting last given command
DAISentOK	last given command has been sent successfully
DAITooManyRetries	last given command could not be sent
DAITransmitIdle	no command given or outstanding

Remarks This function is used to determine if a command has been sent yet, and whether it was successfully sent or not. This function also runs the internal executive which makes the DARTS protocol function. Because of this it should be called "regularly" to allow session to process incoming and outgoing characters.

The function **DAIReadData** also calls the internal executive. Only one of these functions need be called regularly, both can be called.

See Also **DAISendCmd, DAICmdIdle**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
DAITxCmdState CmdStatus;
.
.
/* check on status of any command being sent */
CmdStatus = DAICmdStatus(Session);

/* if not idle and not currently transmitting then must
   have just finished sending a command */
if ((CmdStatus != DAITransmitIdle) &&
    (CmdStatus != DAITxing))
   {
   /* check if failed to send command */
   if (CmdStatus == DAITooManyRetries)
      {
      printf("\n<Unable to send command>\n");
      }

   /* set command status to idle */
   DAICmdIdle(Session);
   }
.
.
```

DAIInit

Description Initialises a created session structure

```
#include <dai.h>

void DAIInit(DAISessionPtr  Session,
             int            PortNumber,
             int            BRate);
```

Session	pointer to DARTS session structure to initialise
PortNumber	port number to use for session
BRate	baud rate setting to use for session

Return Value None.

Remarks Once a communications port has been opened then a session structure needs to be created and initialised. This function will initialise an already created session structure.

The session structure pointer must point to an already defined/allocated session structure in memory

See Also

Example

```
#include <dai.h>
.
.
DAISession ASession;
.
.
/* initialize the session management structure
   use COM1 at 1200 baud */
DAIInit(&ASession,1,1200);
.
.
```

DAINewBaudRate

Description Set new baud rate specific parameters in an existing session structure

```
#include <dai.h>

void DAINewBaudRate(DAISessionPtr  Session,
                    int            BRate);
```

Session	pointer to session structure to set
BRate	baud rate setting session to

Return Value None.

Remarks This function need only be called if the baud rate is changed during a session. This would not normally happen, so this function is rarely used.

The session structure must be created already and initialised.

See Also

Example

```
#include <dai.h>
.
.
DAISession ASession;
.
.
/* Change baud rate for session down to 300 baud */
DAINewBaudRate(&ASession,300);
.
.
```

DAIOpenPort

Description Open a communications port to use for protocoled communications with a Datataker

```
#include <dai.h>

int DAIOpenPort(int  CommPort,
                int  BaudRate);
```

CommPort	communications port number to open (1,2,3,4)
BaudRate	baud rate to open port at (300,600,1200,2400,4800,9600)

Return Value If the communications port is successfully opened then TRUE (1) is returned. If the port number or baud rate is invalid or there is not enough memory left or the port is already open then FALSE (0) is returned.

Remarks This function will open communications port ready for use with a Datataker at the nominated baud rate. The data, stop and parity bits are fixed for the Datataker at 8,1,N respectively. Any port that has been opened should be closed, using the **DAIClosePort** function, before exiting the application or else the interrupt drivers will be undefined and will cause problems. The only supported communications port numbers are 1 to 4 (COM1 to COM4 in PC terminology)

See Also **DAIClosePort**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
/* attempt to open communication port and set nominal
baud rate and line settings */
if (!DAIOpenPort(1,1200))
   {
   printf("\n<Unable to open COM1 at 1200 baud>\n");
   }
.
.
```

DAIReadData

Description Return any data read by DARTS protocol session

```
#include <dai.h>

int DAIReadData(DAISessionPtr   Session,
                char           **BuffPtr,
                int             *BuffLen);
```

Session	pointer to session to receive data from
BuffPtr	returns pointer to a buffer of received data
BuffLen	returns number of characters copied to the buffer

Return Value If any characters are returned in the buffer then TRUE (1) is returned. Else if no characters are returned then FALSE (0) is returned.

Remarks This function will return up to DAI_MAX_DATA_LEN (256) characters in a buffer pointed to by *BuffPtr*.

The caller must process the buffer returned before calling this function again. Else the pointer returned may point to a new message being received, thus losing the last message before it is received. If the message buffer data is required for some time then it should be copied to another local buffer.

The session must have an open port, and be initialised.

This function also runs the internal executive which makes the DARTS protocol function. Because of this it should be called "regularly" to allow session to process incoming and outgoing characters.

The function **DAICmdStatus** also calls the internal executive. Only one of these functions need be called regularly, both can be called.

See Also

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr;
char *Buffer;
int BuffLen;
.
.
/* check if any data has arrived */
if (DAIReadData(Session,&Buffer,&BuffLen))
   {
   /* data has arrived - print it on screen */
   fputs(Buffer,stdout);
   }
.
.
```

DAIReadDirect

Description Reads a character directly from a communcations port for a session, by-passing the DARTS protocol.

```
#include <dai.h>

int DAIReadDirect(DAISessionPtr  Session,
                  char           *ReadChar);,
```

Session	pointer to DARTS session structure to read from
ReadChar	character read from port

Return Value If a character is read then this function will return TRUE (1). Else if no characters are waiting to be read or the port is closed then this function will return FALSE (0).

Remarks The function by-passes the DARTS protocol. It is intended for use to enable the user to establish a connection to a Datataker (NOT for communicating with a Datataker - it can, however, still do it)

See Also **DAISendDirect**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
char ReadChar;
.
.
/* print any characters waiting to be read */
while(DAIReadDirect(Session,&ReadChar))
   {
   printf("%c",ReadChar);
   }
.
.
```

DAISendCmd

Description Specify a command to send via DARTS protocol

```
#include <dai.h>

int DAISendCmd(DAISessionPtr  Session,
               char          *CmdPtr,
               int            CmdLen);
```

Session	pointer to session to send command on
CmdPtr	pointer to buffer containing command to send.
CmdLen	number of characters in command to send.

Return Value If the command is successfully set ready to send then this function will return TRUE (1). However if the last command specified has not yet completed being sent then this function will return FALSE (0) to indicate that this command was not set up and will not be sent.

Remarks This function will set up the command specified ready for transmission. The function **DAICmdStatus** must be called regularly to actually transmit the command and to check when the command has been sent and if it was sent successfully.

The application should call **DAICmdStatus** to determine if the system is still sending the last command before calling this function.

Once the system has read the completion status of the last command to be sent then it can call **DAICmdIdle** to reset the last command status back to DAITransmitIdle.

Commands should include trailing CR (ascii 13) characters if you want the Datataker to process the command.

See Also **DAISendCmdWait, DAICmdStatus, DAICmdIdle**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
.
.
/* send STATUS command to Datataker */
if (!DAISendCmd(Session,"STATUS\x0D",7))
   {
   printf("\n<Couldn't send command,still busy>\n");
   }
while(DAICmdStatus(Session) == DAITxing);
if (DAICmdStatus(Session) == DAITooManyRetries)
   {
   printf("\n<Couldn't send command,excess retries>\n");
   }
.
.
```

DAISendCmdWait

Description Send a DARTS protocoled command to Datataker

```
#include <dai.h>

int DAISendCmdWait(DAISessionPtr  Session,
                   char           *Command);
```

Session	pointer to session management structure
Command	command to send

Return Value if the command is successfully sent then TRUE (1) is returned, otherwise FALSE (0) is returned.

Remarks This function will send a command using the DARTS protocol to the Datataker. It will wait until the command has been sent before returning.

This function **does not** process any input while sending the command. If any input is expected while sending the command then the **DAISendCmd** and **DAICmdStatus** functions should be used to send the command.

The session must be open and initialised and DARTS protocol mode must be on in the Datataker.

Commands should include trailing CR (ascii 13) characters if you want the Datataker to process the command.

See Also **DAISendCmd, DAICmdStatus**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
.
.
/* send test command to Datataker */
if (!DAISendCmdWait(Session,"TEST\x0d"))
   {
   printf("\n<Unable to send TEST command>\n");
   }
.
.
```

DAISendDirect

Description Send a string directly to a communications port for a session, by-passing the DARTS protocol

```
#include <dai.h>

int DAISendDirect(DAISessionPtr  Session,
                  char           *String);
```

Session	pointer to session management structure
String	string to send

Return Value If the string is successfully sent then TRUE (1) is returned. If the port is not open or the output buffer is full or the string could not be sent for some other reason then FALSE (0) is returned.

Remarks This function sends a string directly to the communications port output buffer, by-passing the DARTS protocol.

This function is intended for use to establish a connection to a Datataker. For example to send modem command strings.

See Also **DAIReadDirect**

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
.
.
/* send modem dial command */
if (!DAISendDirect(Session,"ATD123-4567\x0d"))
   {
   printf("\n<Unable to send modem dial command>\n");
   }
.
.
```

DAISetMode

Description Sets DARTS protocol mode on or off on the Datataker

```
#include <dai.h>

int DAISetMode(DAISessionPtr  Session,
               DAIProtMode    Mode);
```

Session	pointer to session structure to use
Mode	mode to set DAIProtocolOff - turns protocol OFF DAIProtocolOn. - turns protocol ON

Return Value If the protocol mode is successfully set then TRUE (1) is returned. If the mode is not set then FALSE (0) is returned.

Remarks This function will attempt to set the Datataker into or out DARTS protocol mode. Three attempts are made to set the requested mode before giving up.

The session must have an open comms port, and be initialised before calling this function.

See Also

Example

```
#include <dai.h>
#include <stdio.h>
.
.
DAISessionPtr Session;
.
.
/* attempt to set the datataker into protocol mode */
if (!DAISetMode(Session,DAIProtocolOn))
   {
   printf("\n<Unable to set DARTS protocol mode on>\n");
   }
.
.
```

Asynch manager interface:

The interface to the Blaise Asynch Manager is contained in the module 'ASY_INTF.C' which can be tailored for any asynch support library. The functions must retain the same parameters and return values, but the actual function code can be changed so as to interface to some other asynch support functions.

Modem connections:

The only direct access provided to the comm port via the DAI functions enables you to send and receive characters. This however is only very basic support. To be able to supervise a modem to dial-up and connect to a Datataker requires some sophisticated functions. The Blaise asynch manager provides such functions. These functions could be called from the application once the comm port has been opened. See the Blaise asynch manager user reference manual for more details - esp. Modem Control Functions.

Mixed language programming:

No special support has been provided for mixed language programming. However this may still be accomplished, but you must be fully aware of the requirements for mixed language programming. See the Language manual's for more information on mixed language programming.