For: karonis
Printed on: Tue, Nov 2, 1993 15:54:19

Document: Channel Access Links
Last saved on: Thu, Sep 3, 1992 15:22:04

Links in a Distributed Database: Theory and Implementation

Nicholas T. Karonis and Martin R. Kraimer

December 1991

1 Introduction

This document addresses the problem of extending database links across Input/Output Controller (IOC) boundaries. It lays a foundation by reviewing the current system and proposing an implementation specification designed to guide all work in this area.

The rest of the document describes an implementation that is less ambitious than our formally stated proposal, one that does not extend the reach of *all* database links across IOC boundaries. Specifically, it introduces an implementation of input and output links and comments on that overall implementation. Finally, we include a set of manual pages describing each of the new functions the implementation provides.

Although we have limited the scope of the implementation by not extending the reach of all database links, the work we have done strictly adheres to the implementation specification formally proposed in section 2. Again, all continuing work in the extension of database links across IOC boundaries should adhere to the same.

2 Theory of Operation

In this section we develop the theoretical framework that serves as a foundation for all work involving the extension of database links across IOC boundaries.

2.1 The Current System

A database is made up of a finite non–empty set of records $\mathfrak{D}$. We can ‘‘connect’’ these records using links; forward–type links (forward links), input–type links (input links), and output–type links (output links). The links in $\mathfrak{D}$ define a binary relation $\mathfrak{L}$, a set of ordered pairs, on set $\mathfrak{D}$. From $\mathfrak{L}$, we can construct a new binary relation $\mathfrak{C}$ on $\mathfrak{D}$ by taking the transitive symmetric reflexive closure of $\mathfrak{L}$. This new binary relation $\mathfrak{C}$ is the equivalence relation induced by $\mathfrak{L}$. Relation $\mathfrak{C}$ partitions the database $\mathfrak{D}$ into a set of equivalence classes. In other words, relation $\mathfrak{C}$ assures that those records that are somehow connected are in the same equivalence class and those records that are not connected are in different equivalence classes.

Currently we load a database, possibly with many equivalence classes, onto a single IOC. We could take a database with multiple equivalence classes and ‘‘split’’ it by loading some of the records onto a different IOC. This could only be done if we were sure that the records we were moving were not ‘‘connected’’ to any of the records that were staying, that is, we never separated records that were in the same equivalence class. If we wanted to split a database, we would be forced to move entire equivalence classes at a time. This restriction on splitting databases, moving only entire equivalence classes at a time, exists because we currently cannot process database links across IOC boundaries. This is something we would like to change.

2.2 Motivation for Something New

Database records are often directly associated with devices and each device is attached to a single IOC. Those database records that are directly associated with a device must reside on the same IOC to which the device is attached.

Frequently, actions taken by one device depend on the state of another device. This interaction is facilitated by the database records associated with each device communicating over database links. This works fine if the two devices are attached to the same IOC, but is currently not possible if they are not.

2.3 Our Proposal

Simply stated, we are proposing to extend the reach of all database links across IOC boundaries. Before stating our proposal more formally, we must first make a distinction in the terminology regarding databases.

When designing a database, it is understood that all the records are stored in a single database file and that all the records in that file reside on a single IOC. We would now like to change that understanding and design databases in which all the records are not necessarily stored in the same database file and, therefore, do not necessarily reside on the same IOC. This introduces ambiguity to the word *database*.

The word *database* could now mean the set of records physically stored in a single database file and placed onto a single IOC, or it can refer to the database as seen by the database designer — a set of records that are connected to each other independent of where each record resides. Until now, those two definitions have always been the same; since there was no independence regarding where records resided, there was no need to distinguish between them.

We call the set of records that are physically stored in a single database file and placed onto a single IOC a *physical* database. The set of records, as viewed by the database designer, that are possibly connected to each other using database links we call a *logical* database.

With that distinction, we are now ready to state the proposal with its implementation specification.

> *We propose to extend the reach of all database links across IOC boundaries. In doing so, we must assure that there is no semantic difference in placing a logical database entirely into one physical database or arbitrarily partitioning it over any number of physical databases.*

By extending the reach of database links we are introducing opportunities for more things to go wrong (e.g., problems associated with the communications hardware and software) that did not exist before. In dealing with these new problems we may be required to make additions to the list of conditions that can raise alarms or even add new alarm types. We are, therefore, creating an environment where a database designer *can* determine whether or not a logical database is partitioned over many physical databases simply by observing alarm conditions that were never possible before. This is *not* a violation of the proposal's implementation specification as it was intended. The proposal's intention is to assure that the semantics of all the parameters specified during database design (e.g., process passive versus not process passive specifications on input and output links) are preserved. It is *not* intended to make the partitioning of logical databases entirely indiscernible.

2.4 Maintaining Loosely Coupled IOCs

In addition to preserving the semantics of all database links, an implementation that extends the reach of database links across IOC boundaries must also strive to keep the IOCs loosely coupled. This constraint is one that is shared by all components of the system.

By its very nature, ''connecting'' IOCs with database links induces tighter coupling between the IOCs. That is unavoidable. However, different implementations might increase the coupling unnecessarily.

It is our goal to minimize the inherent increased coupling effect introduced by any implementation of database links across IOC boundaries. To that end, we designed an implementation that insures that no IOC waits for communication with another IOC by introducing a new locking primitive, ***LOCKNOWAIT***. Additionally, we insure that no IOC is critically dependent on the existence of any other IOC through the use of alarms and a connection handler routine.

3 Input Links

The implementation includes input links (INP, SDIS, DOL, etc.) that are nonprocess passive and either maximize severity or nonmaximize severity. Input links are essentially managed using channel access. A newly–spawned task establishes monitors on the remote source process variables.

3.1 Detecting Links that Refer to Remote Process Variables

During IOC initialization, specifically during database initialization, an attempt is made to resolve the database addresses of the process variables named by the input links. This resolution will only succeed if *both* the source and destination records of the link are in the same physical database. If the resolution fails, it is assumed that the link is referring to a process variable in another physical database.

Each input link that has unresolved database addresses will be represented by a node in a list, the Input List. As IOC initialization continues, specifically during record support initialization, a node is created and added to the Input List for each such link.

3.2 Establishing Monitors on Remote Process Variables

At the end of IOC initialization, a new task is spawned. This new task's responsibility is to establish a monitor on the source process variables represented by each of the elements in the Input List and wait for events to occur. Each monitor is not only responsible for reporting the value of the source field, but also the alarm severity and status of the source record.

3.3 Waiting for Events and Reading Values

After all initialization database processing begins. Each time a monitor reports in, the new value, alarm status, and alarm severity of the source record are copied into the appropriate node in the Input List. Note that the new value is *not* copied into the destination record.

When the record specifying the input link gets processed, the connection to the remote process variable is checked. If the connection has been lost, the destination record's alarm condition is raised to indicate the input value may be invalid. If the connection is intact, the value from the node in the Input List is copied to the destination field. If the input link also requests maximize severity, the destination record's alarm status is also updated using the source record's alarm severity.

4 Output Links

The implementation includes output links that are both nonprocess passive and nonmaximizing severity. Output links are essentially managed by channel access. The output values are written to elements in a shared list. Later a sleeping task awakens, traverses the list, and writes the value to the remote target process variable.

4.1 Detecting Links that Refer to Remote Process Variables

During IOC initialization, specifically during database initialization, an attempt is made to resolve the database addresses of the process variables named by the output links. This resolution will only succeed if *both* the source and destination records of the link are in the same physical database. If the resolution fails, it is assumed that the link is referring to a process variable in another physical database.

Each output link that has unresolved database addresses will be represented by a node in a list, the Output List. As IOC initialization continues, specifically during record support initialization, a node is created and added to the Output List for each such link.

Also during database initialization a shared binary semaphore, ***LookAtBuffer***, is initialized to zero. This semaphore is used for inter task communication and is explained in detail later.

4.2 Establishing Connections to Remote Process Variables

At the end of IOC initialization, a new task is spawned, an output demon. This new task's responsibility is to establish connections and connection handler routines for each of the nodes in the Output List and then wait on the binary semaphore, ***P(LookAtBuffer)***, that is signalled during record processing.

4.3 Writing Values

Each time an output link referring to a remote process variable is processed, the value of the source field in the database record is copied into the appropriate node in the Output List. After the field has been copied, the node in the Output List is moved to the Write List and the binary semaphore is signalled to wake the waiting output demon, ***V(LookAtBuffer)***.

Eventually, the output demon awakens and traverses the Write List. For each node in the list, the connection to the remote destination process variable is checked. If the connection is broken, the source record's alarm condition is raised to indicate the problem. If the connection is intact, the value from the node in the Write List is written to the destination field using channel access and the element is returned to the Output List.

In section 4.2 we noted that a connection handler routine was established for each node in the Output List. The connection handler routine is executed every time the connection status of one of the channels changes. If the connection status goes from disconnected to connected, the connection handler checks if the value associated with the relevant node was *ever* requested to be written. If it was, the value is immediately moved to the Write List (if not already there) and the binary semaphore is signalled, ***V(LookAtBuffer)***. Note that the value is scheduled to be written here even though there was not necessarily a write pending as a result of the associated output link being processed.

As stated previously, we are using one task to copy the database value into the Output List node and then signalling another task, the output demon, to write the value to the remote process variable. We therefore cannot be guaranteed that the output demon will wake up in time to write the value from the list before a new value from the database overwrites it. When processing an output link that refers to a remote process variable, if it is discovered that the new output value is overwriting the previous value that was never sent to the remote process variable, the source record's alarm condition is raised to report the lost value.

4.4 Task Synchronization

The implementation of output links calls for choreography amongst multiple processes. The solution to the problems introduced by a multiple process system is non trivial and, therefore, warrants closer inspection. The implementation for output links calls for three independent processes, possibly executing simultaneously. All three processes interact with the Write List.

4.4.1 Producers and Consumers

Two of the processes are *producers* as they are responsible for placing nodes onto the Write List. The first process executes ***dbcaPutLink()*** during record processing. The second process executes the connection handler routine ***my_connection_handler()*** when the connection status of a channel changes.

The third process is a *consumer* as it is responsible for removing nodes from the Write List. This process is the one introduced in section 4.2. It is the output demon that is explicitly spawned and executes ***dbCaProcessOutlinks()***.

All three processes require mutually exclusive access to the Write List. Mutually exclusive access is provided by the lock ***Buffer***. Each node in the Output List also has its own lock. Additionally, each node can identify whether or not it is on the Write List by inspecting its ***state*** field and determine if its value has ever been requested to be written to a remote process variable by inspecting its ***ever_written*** field.

4.4.2 Locking

In order to maintain loose coupling between IOCs, our implementation introduces a second locking primitive, ***LOCKNOWAIT***, designed to be used with the traditional locking primitive ***LOCK***.

LOCKNOWAIT works as follows. If the lock being sought is available, ***LOCKNOWAIT*** takes the lock and returns immediately with the boolean value ***TRUE***. If the lock is unavailable, ***LOCKNOWAIT*** simply returns immediately with the Boolean value ***FALSE***.

The basic differences between ***LOCK*** and ***LOCKNOWAIT*** are that ***LOCKNOWAIT*** 1) does not wait for a lock it cannot immediately acquire and, 2) returns a Boolean value indicating whether or not it successfully acquired the lock.

This maintains the loose coupling between IOCs by assuring that ***dbCaPutLink()*** never waits for data to be written between IOCs.

4.4.3 The Algorithm

When one of the *producers* places a node onto the Write List, it signals the *consumer* using the binary semaphore introduced in section 4.1, ***V(LookAtBuffer)***.

Initially, each output link that refers to a remote process variable is represented by a node in the Output List and the Write List is empty. The following is the pseudocode for each of the *producers*.

dbCaPutLink()

```
if (LOCKNOWAIT node)
    if (copy value is successful)
        node.ever_written = TRUE
        if (node.state == OFF_WRITELIST)
            node.state = ON_WRITELIST
            LOCK Buffer
            push node onto Write List
            UNLOCK Buffer
            V(LookAtBuffer)
        else
            raise alarm /* overwrite */
        endif
    else
        print error message
    endif
    UNLOCK node
else
    raise alarm /* node already locked */
endif
```

my_connection_handler()

```
if (connection is re-established)
    LOCK node
    if (node.ever_written && node.state == OFF_WRITELIST)
        node.state = ON_WRITELIST
        LOCK Buffer
        push node onto Write List
        UNLOCK Buffer
    endif
    UNLOCK node
    V(LookAtBuffer)
endif
```

The *consumer* is signalled by a *producer* when the *producer* places a node onto the Write List. The *consumer* then wakes up and processes each node in the list.

Each node's connection status is checked. If the connection is intact, the value is written and the node is removed from the Write List. If the connection is broken, an alarm condition is raised and the node is moved to a temporary holding list, the Disconnected List.

Once all the nodes in the Write List have been processed, i.e., returned to the Output List or moved to the Disconnected List, the nodes on the Disconnected List are returned to the Write List. The following is the pseudocode for the *consumer.*

dbCaProcessOutlinks() /* output demon */

```
while (TRUE)
    P(LookAtBuffer)
    done = FALSE
    while (!done)
        LOCK Buffer
        pop node off Write List
        UNLOCK Buffer
        if (node)
            LOCK node
            if (this node’s channel is connected)
                ca_put() node’s value
                state = OFF_WRITELIST
            else
                raise node’s alarm /* disconnected */
                push node onto Disconnected List
            endif
            UNLOCK node
        else
            done = TRUE
        endif
    endwhile
    LOCK Buffer
    Move nodes from Disconnected List onto Write List
    UNLOCK Buffer
endwhile
```

4.4.4 An Optimization

In inspecting the interaction between the first *producer* (the one executing ***dbCaPutLink()*** during record processing) and the *consumer,* we discovered the opportunity for optimization. Consider the following scenario. The system is initialized with an empty Write List. During record processing the first *producer* calls ***dbCaPutLink()***. In doing so, it locks the node, copies the value, moves the node onto the Write List, signals the *consumer,* and unlocks the node.

Eventually the *consumer* wakes up as a result of being signalled by the *producer.* It pops the node off the Write List, locks the node, writes the value, returns the node to the Output List, and unlocks the node.

Consider the amount of context switching that occurs if the *consumer* starts executing immediately after the *producer* signals it. After popping the node off the Write List, the *consumer* attempts to lock the node. This fails because the *producer* still holds the lock. The *producer* must start executing again simply to unlock the node and then the *consumer* can resume execution.

All of this unnecessary context switching can be avoided if the *producer* simply unlocks the node *before* signaling the *consumer.* This is done by moving the unlock node command up into the appropriate places in the ‘‘if’’ structures producing pseudocode that is logically equivalent to the original but is significantly more efficient during execution.

`dbCaPutLink() /* optimized */`

```
if (LOCKNOWAIT node)
    if (copy value is successful)
        node.ever_written = TRUE
        if (node.state == OFF_WRITELIST)
            node.state = ON_WRITELIST
            LOCK Buffer
            push node onto Write List
            UNLOCK Buffer
            UNLOCK node
            V(LookAtBuffer)
        else
            raise alarm /* overwrite */
            UNLOCK node
        endif
    else
        print error message
        UNLOCK node
    endif
else
    raise alarm /* node already locked */
endif
```

This solution to the task synchronization problem ensures that no request will sit on the Write List indefinitely, i.e., the *consumer* will always be signalled to find the node and attempt to write it. It is possible, however, for the *consumer* to wake up and inspect the Write List only to find nothing there.

4.5 Known Problems – Output Links

As mentioned earlier, output links are essentially managed by channel access. The following describes acknowledged problems in the implementation of output links.

4.5.1 Writing Values

Output values are written from the Write List to remote process variables using ***ca_put()***. Calling ***ca_put()*** from a source IOC ultimately results in a call to ***dbPutField()*** on the target IOC. Unfortunately, ***dbPutField()*** writes the value to the field and then decides whether or not to process the target record based on the definition of that field in the destination record's ASCII definition. This is wrong. Whether or not a record gets processed as a result of an output link is determined by the process passive/nonprocess passive specification on the link, *not* the ASCII definition of the target record type.

4.5.2 Connection Management

The destination of the output link has no way of knowing if its value is stale because the source IOC is down. Further, once the source IOC is rebooted, until the record that specifies the output link is processed, the value of the source of the output link and the value of the destination of the output link will be out of synchronization, i.e., the destination will still have the last value received from the last ***ca_put()***.

Currently, it is necessary to spawn a separate vxWorks task from the output demon whose only function is to inherit the output demon's channel access context and make a call to ***ca_pend_event()*** that waits forever. This kludge is the only known way to assure that lost connections are re–established when connecting IOCs using channel access.

5 Comments on the Implementation

In this section we address overall issues associated with the implementation. We discuss the implementation's commitment to loose coupling between IOCs, describe the impact of introducing new database field types, introduce new alarm types, and identify some of the issues and problems that are, as yet, still unresolved.

5.1 Loose Coupling between IOCs

One of the primary goals of the EPICS system is to maintain loose coupling between IOCs. The motivation for this is clear and well–founded; however, the arguments for that decision are beyond the scope of this document.

The implementation of input and output links has strongly adhered to this goal by reading from and writing to buffers that are filled asynchronously. The extent to which tighter coupling exists as a result of the implementation is manifested by the introduction of new alarm types discussed in section 5.3.

5.2 Impact of Adding New Field Types

There should be no need to change any of the code in the implementation as a result of adding new database field types in the future. Provided that new field types are properly handled in channel access (i.e., in the establishment of monitors and in ***ca_put()***), the implementation is entirely immune from any potential anomalies introduced by new database field types.

5.3 New Alarms

As mentioned in section 2.3, by extending the reach of database links we are introducing opportunities for more things to go wrong. In dealing with these new types of errors, we are required to make additions to the conditions that can raise alarms during record processing. The following is a list of conditions introduced by the implementation that are capable of raising alarms.

Severity	Status	Function	Description
VALID	LINK	***dbCaGetLink()***	The monitor channel to the remote source process variable on an input link is discovered disconnected.
VALID	LINK	***dbCaProcessOutlinks()***	The output channel to the remote destination process variable on an output link is discovered to be disconnected by the output demon.
VALID	LINK	***dbCaPutLink()***	The last value to be written was not written in time by the output demon and is being overwritten by the new value.
VALID	LINK	***dbCaPutLink()***	A problem occurred when copying the value from the source process variable to the temporary store for the output demon. The value in the temporary store is labeled invalid instructing the output demon not to send the value, however, the correct value is never sent.
VALID	LINK	***dbCaPutLink()***	The output demon or connection handler had the lock when record processing tried to write another value out on the link. Record processing did not wait and the write was aborted.

5.4 Unresolved Issues – System Wide

This implementation is only intended to implement input and output links that are nonprocess passive and output type links that are nonmaximize severity. The problems of how to implement process passive, output maximize severity links, and forward links are still unsolved.

5.5 Known Problems – Overall Implementation

The following describes the acknowledged problems we encountered in providing the implementation. They are, as yet, still unresolved.

5.5.1 Channel Access and Database Technologies

Channel access has not been updated to understand the new views of databases. Any **.c** file that makes calls to channel access, and hence requires the appropriate channel access header files, cannot include the new database header files. Therefore, we cannot mix channel access and database calls and/or structures in the same **.c** file.

5.5.2 Connection Management

Moving a physical database from one IOC to another will cause all the channel access connections attached to it to be broken and *never* become re–established as long as the database resides on the new IOC. Channel access can only re–establish those channels if the associated process variables, not necessarily the original physical database, are booted onto the original IOC.

6 New Functions

Following is a description of each of the new functions provided by our implementation. Each function is described in a manual page format.

Some of the new functions return values to indicate success or failure. Some of those functions call existing EPICS functions. Occasionally, when functions call existing EPICS functions, their return status is based on the return status received from the existing EPICS functions. For example, ***dbCaAddDbCaInlink()*** calls the already existing ***dbNameToAddr()*** and therefore is capable of returning any return code that ***dbNameToAddr()*** can return, including its return code to indicate successful completion.

Throughout the following documentation we have adopted a naming convention regarding arguments that refer to database field types. The convention is as follows:

{dest, source}_{new, old}_{dbf, dbr}_type

where

dest – a description of a destination database field type
source – a description of a source database field type

new – a description using the *new* database field types
old – a description using the *old* database field types

dbf – a description using the DBF_XXX field type
dbr – a description using the DBR_XXX field type

NAME

dbCaAddInlink() – called for each input PV_LINK during record initialization

SYNOPSIS

```
long dbCaAddInlink(plink, pdest_record, dest_fieldname)
      struct link *plink;
      void *pdest_record;
      char *dest_fieldname;
```

DESCRIPTION

This routine is called during record initialization for each input link that is still PV_LINK, i.e., those links that failed to resolve database addresses during database initialization and, hence, be converted from PV_LINK to DB_LINK. It finds the database address of the destination process variable specified by ***pdest_record*** and ***dest_fieldname*** by calling ***dbNameToAddr()*** and dynamically allocates the appropriate number of bytes as a temporary store for incoming values from channel access monitors. The link type is converted from PV_LINK to CA_LINK.

RETURNS

any return code from ***dbNameToAddr()*** (including its success rc)

S_dbca_failedmalloc	unable to dynamically allocate memory
S_dbca_nullarg	one of the incoming pointer arguments was NULL
S_dbca_dbfailure	problem in calling one of the db routines

SEE ALSO

dbCaGetLink() and ***dbCaProcessInlinks()***

NAME

dbCaAddOutlink() – called for each output PV_LINK during record initialization

SYNOPSIS

```
long dbCaAddOutlink(plink, psource_record, source_fieldname)
      srtuct link *plink;
      void *psource_record;
      char *source_fieldname;
```

DESCRIPTION

This routine is called once during record initialization for each output link that is still a PV_LINK, i.e. those links that failed to resolve database addresses during database initialization and, hence, be converted from PV_LINK to DB_LINK. It finds the database address for the source process variable specified by ***psource_record*** and ***source_fieldname*** by calling ***dbNameToAddr()*** also dynamically allocates the appropriate number of bytes memory to act as a temporary store between the time the record is processed and the time it is sent to the remote process variable. The link type is converted from PV_LINK to CA_LINK.

RETURNS

any return code from ***dbNameToAddr()*** (including its success rc)

S_dbca_failedmalloc	unable to dynamically allocate memory
S_dbca_nullarg	one of the incoming pointer arguments was NULL
S_dbca_dbfailure	problem in calling one of the db routines

SEE ALSO

dbCaPutLink() and ***dbCaProcessOutlinks()***

NAME

dbCaGetLink() – called by record processing when inputting on a CA_LINK

SYNOPSIS

```
long dbCaGetLink(plink)
      struct link *plink;
```

DESCRIPTION

This routine is called during record processing when reading from an input CA_LINK. If the connection to the source process variable is noted to be down when this routine gets executed, then no value is read and an alarm condition is raised in the destination record. If the connection is intact, then the value is copied from the temporary store acquired by ***dbCaAddInlink()*** by calling ***dbPut()***.

RETURNS

any return code from ***dbPut()*** (including its success rc)

S_dbca_nullarg	one of the incoming pointer arguments was NULL
S_dbca_foundnull	found NULL pointer where one should not be

SEE ALSO

dbCaAddInlink() and ***dbCaProcessInlinks()***

NAME

dbCaLinkInit() – called during IOC initialization

SYNOPSIS

```
void dbCaLinkInit(count)
      int count;
```

DESCRIPTION

This routine is called twice during IOC initialization, once before record support initialization to initialize variables and once after record support initialization to spawn necessary tasks. The argument ***count*** must be one for the first call and two for the second call.

RETURNS

N/A.

SEE ALSO

None.

NAME

dbCaProcessInlinks() – *spawned* once after all record initialization

SYNOPSIS

```
void dbCaProcessInlinks()
```

DESCRIPTION

This routine is spawned as a separate task at the end of IOC initialization. If there are no input CA_LINKs in the database, this task terminates immediately. If there are input CA_LINKs, this routine initiates the search for the remote source process variables and establishes monitors on them specifying our own event handler routine. It then waits forever for incoming events on those monitors.

Our event handler routine copies the data value and the alarm status from the remote source process variable. Later, when processing the record with the input link, the value and alarm status of the source process variable are read from the copied values.

RETURNS

N/A.

SEE ALSO

dbCaAddInlink() and ***dbCaGetLink()***

NAME

dbCaProcessOutlinks() **–** *spawned* once after all record initialization

SYNOPSIS

```
void dbCaProcessOutlinks()
```

DESCRIPTION

This routine is spawned as a separate task at the end of IOC initialization. It is the output demon. If there are no output CA_LINKs in the database, this task terminates immediately. If there are output CA_LINKs, this routine initiates the search for the remote process variables and provides a connection handler routine.

This routine waits for requests to output values to remote destination process variables. As it receives each request, the connection status to the remote destination process variable is checked. If the connection is intact, the value is written. If the connection has been broken, an alarm condition is raised in the source record.

The connection handler routine is responsible for detecting changes in connection status and processing those connections whose status goes from disconnected to connected. For those connections, if a request to write the value has ever been made, the value is immediately scheduled to be written again.

RETURNS

N/A.

SEE ALSO

dbCaAddOutlink() and ***dbCaPutLink()***

NAME

dbCaPutLink() – called by record processing when outputting on a CA_LINK

SYNOPSIS

```
long dbCaPutLink(plink, poptions, pnrequest)
      struct link *plink;
      long *poptions;
      long *pnrequest;
```

DESCRIPTION

This routine is called during record processing when writing to an output CA_LINK. The value to be written is copied to a shared buffer using the database routine ***dbGetField()***, hence the need for the two arguments ***poptions*** and ***pnrequest***.

After the value is copied, this routine determines if there is a pending output request on this process variable. If there is, an alarm is raised to indicate that an output value has been lost (not outputted) and a new value has been written over it. If there is not, another task, the output demon, is signaled to wake up and write the value.

RETURNS

any return code from ***dbGetField()*** (including its success rc)

0	successfully raised alarm status
S_dbca_nullarg	one of the incoming pointer arguments was NULL
S_dbca_foundnull	encountered a NULL pointer where one should not be
S_dbca_dbfailure	problem in calling one of the db routines

SEE ALSO

dbCaAddOutlink() and ***dbCaProcessOutlinks()***