Title: Successful Business Continuity – Part 1 of 5

During a Disaster Recovery (DR) implementation effort the last thing you want is unexpected hardware and software configuration issues. These tend to consume time, resources, and cause Recovery Time Objectives (RTO) to be missed. In order to ensure business continuity, organizations must design, implement, maintain and enforce policies, procedures, standards, and guidelines that encompass all aspects of their critical business functions.

A successful DR effort is not only dependent upon a well thought-out DR plan; it must have been derived from an enterprise wide mentality of business continuity. Furthermore, business continuity must be the beginning point in systems design, not the end point. Unfortunately, very few systems are built from the business continuity perspective backwards.

This series of articles will discuss AIX/HACMP systems design from a business continuity perspective, and will provide scripting examples to support the suggested policies, guidelines, standards, and procedures.

Business Continuity is the activity performed by an organization to ensure that critical business functions will be available to customers, suppliers, regulators, and other entities that must have access to those functions. These activities include many daily chores such as project management, system backups, change control, and help desk. Business Continuity is not something implemented at the time of a disaster; Business Continuity is those activities performed daily to maintain service, consistency, and recoverability.

The foundation of Business Continuity is the policies, guidelines, standards, and procedures implemented by an organization. All system design, implementation, support, and maintenance must be based on this foundation in order to have any hope of achieving Business Continuity, Disaster Recovery, or in some cases, system support.

This series of articles will define many IT areas that require enterprise wide policies, guidelines, standards, and procedures be defined, and will offer recommended solutions for those defined areas. The areas discussed will include:

- Article 1:
  - User Names and UID Numbers
  - Group Names and GID Numbers

- Article 2:
  - Machine names
  - Hostnames
  - Boot adapter and service names
  - Resource group names
  - Aliases

- Article 3:
  - Volume Groups
  - Major Numbers
  - Logical Volumes
  - JFS Log Logical volume names
  - Mount points

- Article 4:
  - MQ Series Queue names and aliases
  - Resource Group start/stop scripts
  - Error logging
  - Error Notification

- Article 5:
  - Automated Documentation
  - Console Access
  - Job Scheduling
  - Project Planning

It is usually a bad idea to attempt to maintain multiple standards in support of a specific IT area. In many organizations today, separate standards are maintained for standalone AIX systems and HACMP clusters. This series of articles will show that is not necessary and how to consolidate into a single standard for all AIX systems, including HACMP.

Definition: Enterprise wide unique - refers to a parameter that has one distinct value across any or all platforms throughout the entire enterprise.

User and Group Names - UID/GID Numbers

On an AIX or Unix system, files are not stored by filename, they are stored by "inode number". Each file has an inode and is identified by an inode number, sometimes called an i-number in the file system where it resides (*Reference 1*). inodes provide important information on files such as user and group ownership, access permissions and file type. Each file on a Unix system is associated with exactly one user (owner) and one group.

Reference 1: http://www.webopedia.com/TERM/I/inode.html

Similar to the filename/inode relationship, the file owner and group membership is designated and stored by the UID/GID numbers. Each user on a Unix system is assigned a UID number and each group a GID number. These numbers are used by the inode to assign ownership and group membership to files.

In Business Continuity Planning, it is important to recognize that UID/GID numbers should be uniquely assigned to users and groups on an enterprise wide basis.

If these UID/GID numbers are not enterprise wide unique, there will be security issues as well as conflicts during high availability or disaster recovery failovers. Security issues will also arise when a backup is restored onto a machine where the UID/GID numbers do not match those stored in the backup.

The following are recommended policies, standards, guidelines, and procedures, as related to "User and Group Names - UID/GID Numbers":

Policies: User and Group Names - UID/GID Numbers

Each user provided access to any system shall have an enterprise wide unique username assigned to them. This username will be implemented on each system to which this user requires access.

Each user provided access to any system shall have an enterprise wide unique UID number assigned to them. This UID number will be implemented on all systems requiring a username/UID association.

Each group name implemented on a Unix system shall have an enterprise wide unique GID number assigned to it. This GID number will be implemented on every system where this group name is implemented.

Guidelines: User and Group Names - UID/GID Numbers

It is important to choose a format for user names that provides enterprise wide flexibility for use with as many platforms as possible. A user name format that appears to be supported on the widest range of platforms (AIX, Linux, AS/400, MVS, VSE, MS Windows, Unix) is a seven character username, the first three characters being lowercase letters, the last four characters being digits from 0 - 9.

In order to avoid the maintenance and support issues of keeping a database of UID/GID values and their associated user and group names, a reproducible algorithm should be used to calculate the UID/GID values. This algorithm should be executable on any Unix system and provide the same UID/GID number for a given user or group name. Since the UID/GID numbers are reproducible on any Unix platform, the only values that must be maintained are the enterprise wide unique user and group names.

Standards: User Names and UID Numbers

In order to facilitate normal maintenance, disaster recovery and business continuity, it is recommended that each enterprise wide unique user and group name, also be assigned enterprise wide unique UID or GID number.

This will cause files whose owners/groups are not defined in the /etc/password file to appear in a long listing with the enterprise wide unique UID and GID numbers for the owner and group. This condition may indicate which users need to be added to the system, or which files need to be removed from the system. Regardless of the fix, file security will remain intact and uncompromised.

Instead of keeping a database of username/UID and group name/GID associations, it is preferred to use an algorithm to generate the UID/GID values based on the username/group name. This algorithm must be reproducible across all systems that require a UID/GID number.

A typical algorithm for performing a UID/GID calculation is to use the "sum" command with the "-r" option to generate the Berkeley cksum value. The "-r" option is supported by AIX and appears to be consistently implemented across a wide variety of other Unix platforms as well. An example of using this technique (all commands shown in this series of articles are expressed in Korn shell syntax):

```
$ print "abc1234" | sum -r
29247  1
```


A limitation of this technique is that it will only calculate numbers between about 600 and 65,000 for the specified username format of three lowercase letters followed by 4 digits. So the number of users and groups, enterprise wide, is limited to less than 65,000, and the possibility of duplicates does exist with this algorithm. These limitations may be acceptable for some organizations; however most will want to expand this concept of UID/GID calculation.

The recommended UID/GID calculation algorithm uses a base 26 numbering system to represent the first three characters of the username, and then converts the calculated base 26 number to a base 10 (decimal) value. An example script implementing this technique is shown below.

---- Cut Here ----

```
#!/usr/bin/ksh93
#####################################################################
function usagemsg_mkuid {
  print "
Program: mkuid

This function generates a UID number for a username,
that username must consist of 3 letters followed by 4
digits,  or 4 letters followed by 3 digits.  The
username is assumed  to be the first argument to the
function.  The username must  also be exactly 7
characters long.

Usage: ${1##*/} username
  Where:
    username = XXX#### - 3 letters followed by 4 digits
       Generates UID between 1,000,000 and 176,759,999
            or
    username = XXXX### - 4 letters followed by 3 digits
       Generates UID between 176,760,000 and 633,735,000

Author: Dana French (dfrench@mtxia.com) Copyright 2004
\"AutoContent\" enabled
"
}
#####################################################################
####
#### Description:
####
#### This function generates a UID number for a username, that
#### username must consist of 3 letters followed by 4 digits,
#### or 4 letters followed by 3 digits.  The username is assumed
#### to be the first argument to the function.
####
#### Assumptions:
####
#### This function assumes the username is constructed with
#### either 3 or 4 contiguous alphabetic characters followed
```

```
#### by 4 or 3 contiguous digits.  The total number of
#### characters in the user name must be exactly 7.
####
#### Dependencies:
####
#### The "mkuid" function is dependent upon the external
#### function "mkascii" to produce an associative array
#### of ascii decimal values for the lower case letters.
####
#### Products:
####
#### Upon successful completion this function prints a single
#### decimal value to standard output.
####
#### For usernames consisting of 3 lower case letters
#### followed by 4 digits, the resulting UID values are
#### between 1,000,000 and 176,759,999.
####
#### For usernames consisting of 4 lower case letters
#### followed by 3 digits, the resulting UID values are
#### between 176,760,000 and 633,735,000.
####
#### Configured Usage:
####
#### The "mkuid" function can be called from the command
#### line, script, or another function.
####
#### Details:
####
#########################################################################
function mkuid
{
  typeset -l LET
  typeset NUMLET=0

  [[ "${1}" == "-?" ]] && usagemsg_mkuid "${0}" && return 1

####
#### If the total number of characters in the username
#### command line argument is not equal to 7, display an
#### error message, followed by the usage message, and return
#### from the function with an unsuccessful return code.
####

  if (( ${#1} != 7 ))
  then
     print -u 2 "ERROR: Number of characters in username ${1} is not equal to 7\n"
     usagemsg_mkuid "${0}"
     return -1
  fi

####
#### Extract only the alphabetic characters from the
#### username command line argument and count the number
#### of characters extracted.
####
```

```
 NUMLET="${1//([!a-zA-Z])/}"
 NUMLET="${#NUMLET}"

####
#### If the number of alphabetic characters in the username
#### command line argument is not equal to 3 or 4, display an
#### error message, followed by the usage message, and return
#### from the function with an unsuccessful return code.
####

 if (( NUMLET != 3 )) && (( NUMLET != 4 ))
 then
    print -u 2 "ERROR: Number of letters in username ${1} is not equal to 3 or 4\n"
    usagemsg_mkuid "${0}"
    return -1
 fi

#### Determine the base 10 order of magnitude for the
#### numeric portion of the user name and store this value.

 (( ORDMAG = 10 ** ( 7 - NUMLET ) ))

#### Initialize the decimal value of the lower limit for the
#### UID number to one million.  The lowest value allowed for
#### a calculated UID number will be this value.

 typeset LOWLIM=1000000

#### Initialize the decimal value of the UID number using a
#### base 26 calculation.  The lower limit is added to this
#### value to insure the calculated UID number is greater
#### than the lower limit.

 (( LOWUID = ( 26 ** ( NUMLET - 1 ) * LOWLIM / 100 ) + LOWLIM ))
 (( NUMLET == 3 )) && (( LOWUID = LOWLIM ))

#### Initialize several variables containing values used
#### while iterating through each character of the username.

 typeset BASE=26
 typeset NVALUE=""
 typeset LVALUE=0
 typeset BASEORDMAG=0
 typeset SUBTOT=0
 typeset TOT=0
 typeset MULT=0

####
#### Define an associative array to contain an ascii table of
#### values, the array index is the alphabetic character, the
#### value is the decimal number associated with the
#### character.  Call the external function "mkascii" to
#### create this array.
####

 typeset -A LETTERS
 mkascii LETTERS
```

```
#   print " ${LETTERS[@]}"

####
#### Divide the username into individual characters and store
#### each character in an array.  Iterate thru this array one
#### element at a time to calculate the base 26 value of each
#### alphabetic character and the decimal value of each
#### number.
####

  eval EACHLET="( ${1//(?)/ \1} )"
  for LET in "${EACHLET[@]}"
  do

####
#### If the current interation character is a lower case
#### letter, subtract the decimal value of 97 from the ascii
#### value for this character ( 97 is the ascii value of the
#### lowercase letter "a" ).  This will cause the value of
#### "a" to be zero, "b" = 1, "c" = 2, etc.  Assign a
#### variable to contain this decimal value for the letter.
####

    if [[ "_${LET}" = _[a-z] ]]
    then
      LVALUE="$(( ${LETTERS[${LET}]} - 97 ))"

####
#### If the current iteration character is not a lower case
#### letter, then it is a number from 0 - 9.  Set the letter
#### value to zero (indicating the iteration character is not
#### a letter) and append the number to the end of the
#### numeric value.
####

    else
      LVALUE="0"
      NVALUE="${NVALUE}${LET}"
    fi

####
#### Calculate a base 26 multiplier for the current iteration
#### of the loop.  Each loop iteration should increase this
#### multiplier by a base 26 order of magnitude.
####

    (( MULT = BASE ** BASEORDMAG ))

####
#### Multiply the decimal letter value by the base 26
#### multiplier and add the product to a running total for
#### each iteration of the loop.
####

    (( SUBTOT = SUBTOT + ( LVALUE * MULT ) ))
```



```
####
#### Add a value of 1 to the loop counter, which is used to
#### determine the base 26 order of magnitude.
####

   (( BASEORDMAG = BASEORDMAG + 1 ))

  done

####
#### Multiply the value calculated for the alphabetic
#### characters by the base 10 order of magnitude for the
#### numeric portion of the user name.  Add the numeric
#### portion of the user name and the lower limit value to
#### arrive at a final value for the calculated UID number.
#### Print this value to standard output and return from this
#### function with a successful return code.
####

  (( TOT = ( ( SUBTOT * ORDMAG ) + NVALUE ) + LOWUID ))
  print ${TOT}
  return 0
}
#########################################################################
function usagemsg_mkascii {
  print "
Program: mkascii

This function accepts an associative array variable name
as the first command line argument, and builds an ASCII
table of characters in that array.  The index of the
associative array is the ASCII character.  The value
contained in each array element is the decimal, hex, or
octal number associated with the ASCII character array
index.

Usage: ${1##*/} ArrayName
  Where:
    ArrayName = Name of a predefined associative array

Author: Dana French (dfrench@mtxia.com) Copyright 2004
\"AutoContent\" enabled
"
}
#########################################################################
####
#### Description:
####
#### The "mkascii" function returns an ASCII table of
#### characters in an associative array. The index of the
#### associative array is the ASCII character.  The value
#### contained in each array element is the decimal, hex, or
#### octal number associated with the ASCII character array
#### index.
####
#### The "mkascii" function will accept one or two command
#### line arguments.  The first is the name of a predefined
```

```
#### associative array, the optional second command line
#### argument can be specified to designate the type of value
#### (decimal, octal, hexidecimal) to return in the
#### associative array.  Decimal values are returned
#### default.  If the value of the second command line
#### argument is "8", octal values are returned, if the value
#### of the second command line argument is "16", hexidecimal
#### values are returned.
####
#### Assumptions:
####
#### It is assumed the first command line argument is the
#### name of a predefined associative array.  If the second
#### command line argument is not specified, it is assumed
#### the user is requesting decimal values be returned in the
#### array.
####
#### Dependencies:
####
#### The only valid values for the second command line
#### argument are "8" (octal)  and "16" (hexidecimal).
#### Any other values for the second command line argument
#### will result in an error.
####
#### Products:
####
#### The product of the "mkascii" function is the completion
#### of a predefined associative array indexed by the ASCII
#### character whose value is the decimal, octal, or
#### hexidecimal number that represents the ASCII character.
####
#### Configured Usage:
####
#### The "mkascii" function can be called from the command
#### line, script, or another function.
####
#### Details:
####
########################################################################
function mkascii
{

####
#### If the first command line argument is a literal "-?",
#### display the usage message and exit the function with an
#### unsuccessful return code.
####

  [[ "${1}" == "-?" ]] && usagemsg_mkascii "${0}" && return 1

####
#### Establish a value variable to contain a decimal value associated
#### with each ASCII character.  The variable containing the
#### decimal value is unevaluated because the evaluation will
#### take place later.
####
```

```
  typeset VAL='${CNT}'

####
#### If the second command line argument is "8", change the
#### value variable to contain a statement that will be
#### evaluated later to octal values.
####

  [[ "_${2}" != "_" && "_${2}" =  "_8" ]] && VAL='$( printf "%o" 0x${i}${j} )'

####
#### If the second command line argument is "16", change the
#### value variable to contain a statement that will be
#### evaluated later to hexidecimal values.
####

  [[ "_${2}" != "_" && "_${2}" = "_16" ]] && VAL='${i}${j}'

####
#### Using the first command line argument as the name of a predefined
#### associative array, create a name reference to that array.
####

  nameref ASCII_TABLE="${1}"

####
#### Initialize a loop counter incremented by one each time
#### through the loop.  This counter is used to represent the
#### decimal value of the ascii character.
####

  typeset CNT=0

####
#### Loop through the double digit hexidecimal values for all
#### ASCII characters.
####

  for i in 0 1 2 3 4 5 6 7 8 9 A B C D E F
  do
    for j in 0 1 2 3 4 5 6 7 8 9 A B C D E F
    do

####
#### Evaluate the ASCII character from the double digit
#### hexidecimal value for the current iteration of the
#### loops.  Also evaluate the decimal, octal or hexidecimal
#### value associated with the ASCII character.  Assign
#### the resulting value to the associative array using the
#### "nameref" variable.
####

      eval ASCII_TABLE[\\$(print -- $( printf "\\\0%o" 0x${i}${j} ) )]=\"${VAL}\" 2>/dev/null

####
#### Increment the loop counter by one (used to represent
#### decimal values of each ASCII character
```

```
####

    (( CNT = CNT + 1 ))

####
#### Return to the beginning of the loop to evaluate the next
#### character in the ASCII sequence.
####

   done
  done
}
#########################################################################
####
#### Call the "mkuid" function with all command line arguments

mkuid "${@}"
```

---- Cut Here ----

In the “usage message” areas of the above script, the phrase “AutoContent Enabled” appears. This refers to a technique of commenting scripts in such a way that documentation can be automatically generated from the script. This has the added benefit that whenever updates to the script are made, the documentation is automatically updated also. Notice that all comments in the script begin with 4 hash marks (#) followed by a space, this pattern is used to designate text used as documentation. This automated documentation technique, referred to here as “AutoContent”, will be discussed in a later article.

The “mkuid” script shown above is composed of shell functions so that, if desired, the script can be separated into individual components and called from a shell function library. Otherwise, it can be executed as-is from the command line.

Standards: Group Names and GID Numbers

Since the group names will likely not conform to the same standard selected for the usernames, a different mechanism is required to provide the group name/GID number associations. Software vendors, suppliers, and internal operations will specify group names that must be implemented as-is. For instance the Informix database will require the group name "informix" be implemented on each system running the database.

The recommended UID calculation method which implements a base 26 conversion of the username to a decimal value cannot be used in this instance, because it will generate decimal values greater than that supported by AIX and/or HACMP. Therefore and alternate method is needed for GID calculations. The previously

mentioned Berkeley "sum -r" method will likely suffice for this purpose. The limitation of 65,536 names does not cause a problem in reference to defining group names, because it is extremely unlikely that this limitation would ever be reached.

So the recommended algorithm to use for defining the group name/GID number associations is the Berkeley "sum -r" method:

```
$ print "informix" | sum -r
21883  1
```

Whatever unique UID/GID calculation algorithm is selected, chosen, or written, it is important that it be implemented enterprise wide to eliminate security issues and failover conflicts during high availability or disaster recovery events. This technique also provides added security during normal maintenance activities by insuring files are not accidentally assigned to a user or group that does not own the file.

Procedures: User and Group Names - UID/GID Numbers

As part of the standard build for each AIX system, copy the "mkuid" script to each system, the recommended location is:

```
/usr/local/sh/mkuid
```

When implementing a new user on an AIX system thru the "SMIT" interface, the UID number should first be calculated. An example is shown for a username "abc1234":

```
/usr/local/sh/mkuid abc1234
```

The resulting number should be used as the "User ID" value when adding the user "abc1234" on an AIX system thru the "SMIT" interface.

When implementing a new user thru a script, the resulting number from the "mkuid" calculation should be captured into a variable and used as the UID number.

When implementing a new group on an AIX system thru the "SMIT" interface, the GID number should first be calculated using the Berkeley "sum -r" method. An example is shown for a group name "informix":

```
$ print "informix" | sum -r | awk '{ print $1 }'
```


21883

The value "21883" should be used as the Group ID number when adding the group "informix" on an AIX system thru the "SMIT" interface.

When implementing a new group thru a script, the resulting number from the "sum -r" calculation should be captured into a variable and used as the GID number.

Conclusion:

The result of standardizing user names, group names, UID numbers, and GID numbers includes the following:

- Individual login names for each user
- Capable of centralized user/group management
- Increased security
- Improved auditing capabilities
- Avoidance of user name conflicts
- Avoidance of group name conflicts
- Avoidance of UID conflicts
- Avoidance of GID conflicts
- Avoidance of Security holes during and after DR

Conflict avoidance is important during a disaster recovery effort because conflicts tend to consume large amounts of time. The last place you want to redesign and implement new user/group standards are when you are at your disaster recovery site attempting to recover your production systems. For most organizations, their disaster recovery provisions are not an exact duplicate of their production systems. In a disaster recovery implementation, multiple systems may be consolidated onto a single platform. This consolidation will expose conflicts such as user/group names and ID numbers, and will require these conflicts be resolved before the production systems can be recovered.

For those organizations where the disaster recovery provisions are an exact duplicate of their production systems, standardization of user/group names and ID numbers using this technique, is also highly desirable for the purpose of simplifying user/group support, maintenance and security. Again, file security on AIX is enforced by UID/GID numbers, not user/group names; therefore the production and disaster recovery systems must be synchronized on this aspect.

The best way to avoid conflicts during disaster recovery is to implement and enforce policies, guidelines, standards, and procedures to eliminate conflicts as part of your enterprise wide business continuity planning.

The next article in this series will discuss naming structures for machines, hosts, adapters, resource groups, and aliases for use in a business continuity environment.

