Subsystems - Overview

- Three major functional areas:
  - — Views and view management; releases, specs, etc.
  - — Check-in/Check-out; reservations
  - — Work orders

Subsystems - Definitions and Terms

- View - A version of a subsystem. Contains the Ada units and files for a particular version of the entire subsystem.

- Configuration - A configuration is a handle to the values of controlled objects in the source database. In the Configurations directory is additional information that allows one to recreate a view from a configuration. controlled objects only

- Working view - A view that is under development. Name ends in "_working".

- Release view - A view that has been "released". It is frozen and cannot be modified. Name does not end in "working".

Definitions and Terms cont'd...

- Spec view - A view of a subsystem that contains only specs that can be referenced for execution or imported by other subsystem views.

- Element - An object in a view or views. An object with the same name in two views is the same element.

- Controlled Object - An object that is under source control.

  — Requires check-in/check-out for editing

  — 255 character line length limit

  — No non-ascii characaters

  — Information on all changes available

  — Problems with very large files/units

Definitions and Terms cont'd...

- Joined Objects/Severed Objects - If objects that are the same element share a reservation token, they are said to be joined. Otherwise, they are severed.

Subsystem - Characteristics

- Set of views
  - — These appear in the subsystem root directory.
- Configurations
  - — These appear in a configurations directory in the subsystem root. Other files in the configurations directory specify imports and other information needed to build a view that corresponds to a configuration.
- CMVC Database
  - — This appears in the State directory in the root of the subsystem. It contains source information and other information about views and controlled objects in the subsystem.
- Compatibility World
  - — This world contains the information needed to guarantee that code is compatible in the face of incremental changes, code views, and multi-machine development.

Subsystem - View Characteristics

- Units - Contains files and Ada units of the subsyste m view. No Worlds - causes problems
- State - Contains control and management information.
- Imports - Imports directory contains import restriction files.
- Exports - Exports directory contains export subset files. Spec. view
- Tool_State - Subdirectory of State; copied when view is copied. used by Target. Builder, etc.
- Model- Provides base of links and switches. Can change in a view, may require demotion
- Imports - Represents imported subsystem spec views. Not editable.
- Switches - Initially copied from Model, but copy lives in state directory.

- Referencers - Says which subsystem views import this one. Doesn't archive from machine to machine too well, as the views named in referencers may not exist on the other machine. CMVC Check_Consistency will fix any problems in Referencers.

Used for Import validation

Making New Views

- Copy
  - — Makes spec and working views
  - — Level and Naming - Use Natural'last to set name explicitly.
- Make_Spec_View
- Release

Imports and Model

- Import
- Remove_Import
- Remove_Unused_Imports
- Replace_Model

Deleting

- Destroy_View
- Destroy_Subsystem

must be empty, i.e. no views

to destroy config (after Destroy_View but config.also => false
keeping cfg allows view to be recreated
- delete config object from configuration directory
- run cmvc_maintenance.Expunge_Database

Re-Creating Views

- Build

CMVC Source Control

- Provides Control Over Changes
  Changes can only be made between CHECK_OUT and CHECK_IN

- Change History Is Automatically Kept
  CMVC keeps line differentials of changes to text/source.

- Provision For Simultaneous Development
  Multiple developers can work without losing changes or getting in each other's way.

- Provision For Multi Release/Target Development
  Changes can proceed independently on two targets or releases, and be merged later if desired.

Terminology

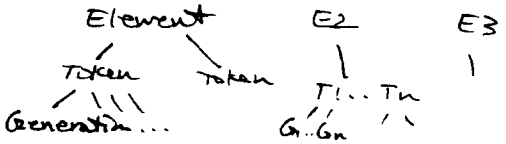
- Controlled Object
  An object is *Controlled* if it is known to CMVC source management. A controlled object must be checked out before it can be changed. Only Ada and Text objects can be controlled. cannot be edited, parsed (TEXT_IO, etc) or deleted

- Element
  An element is the set of all objects in a subsystem that have the same name from the units directory down to the object. For example, the two objects `SS.Rev9_Working.Units.Foo` and `SS.Rev9_Cbh_Working.Units.Foo` are in the same element.

- Reservations
  A controlled object is *Reserved* if it is checked out.

- Reservation Token
  Every controlled object has a reservation token associated with it. Reserving an object is accomplished by getting and holding its reservation token with the object.

  Objects in the same *Element* can share the same reservation token. The effect is that when any one object in the element has the reservation token, no other object can be checked out. Objects that share a reservation token are said to be *Joined.*



- GENERATION
  each check out / check in sequence produces a new generation
  history is all changes between, coalesced into one "change"

Terminology - Cont.

- Path
  A path is a series of releases, ending in a working view.

  A subsystem can contain many paths. Objects in two paths may or may not be joined.

- Subpath
  A subpath is a (possible) set of releases, and a working view. It is related to its path in that all of the objects in the subpath are joined with the those in the path.

- Check_Out
  CHECK_OUT searches for the reservation token associated with the object. If the token isn't currently attached to some other object, it is attached to the one named in the CHECK_OUT command, and the object is made editable.

  A CHECK_OUT might imply an ACCEPT_CHANGES; see below.

Terminology - Cont.

- Check_In

  CHECK_IN releases the reservation token. It then computes the difference between what the object used to look like and what it looks like now and stores these differences in the Cmvc Source Database.

- Accept_Changes

  ACCEPT_CHANGES looks at all of the objects that share a reservation token and finds the latest set of changes. These changes are copied to the object named in the ACCEPT_CHANGES command.

  Since CHECK_OUT allows an object to be changes, it must be brought up to date before changes can be made. Thus CHECK_OUT might perform an ACCEPT_CHANGES as part of its operation.

Configurations

- Represents a Snapshot Of Controlled Objects
  The configuration is a handle into the CMVC Source Database. It allows retrieval of the text for controlled objects. Since there is a configuration for every view, past and present, they allow reconstruction from history of old views.

  They also allow queries regarding what has changed over time, for example between two views, between two configurations, or some combination.

- Configurations Require Minimum Storage
  They require almost no storage beyond that required anyway to keep source history.

- Deleting A View Doesn't Delete Its Configuration
  Unless specifically requested, the configuration will remain behind (in the subsystem Configurations directory), allowing reconstruction of the view using BUILD.

Relocation

- Copies Installed And Coded Ada Units
  The relocation process copies Ada units in the installed or coded state. This saves compile time. For example, the Cmvc source consists of 140 units, 17.5 KSLOC, and requires the following time to compile/relocate (using D_9_21_0).

Time in Minutes		
Goal State	Using Compilation	Using Relocation
Source Copy	9	-
Installed	30	19 (Includes Copy)
Coded	43	21

- Used By COPY And Release
  Relocation is used to create the new views if views are to be created.

- Used By ACCEPT_CHANGES To Assist Copies
  ACCEPT_CHANGES tries to relocate units in the installed state if possible. It isn't possible if the source isn't installed, or if the destination hasn't been relocated from the source. This is very complicated.

Forms Of Accept Changes

- Object To View
  Copies the specified object to the specified view, if needed. If the object doesn't exist in the view, create it and make it controlled with the same reservation token as the source.

- Object To Object
  Copies the specfied source object over the destination object if needed. Both objects must be in the same element. not very common

- View To View
  Copies all changed objects from the source view to the destination. New objects in the source are added to the destination. This will not revert an object; i.e. its generation number will never back up. will try to make the views relocation equivalent could take a long time

- Latest to Object
  Find the latest version of the object in any view, and copy it to the destination.

- Latest to View
  Perform Latest to Object on every object in the view. This will never revert.

- View To Configuration
  Make the view look like the objects specified in the source configuration. This WILL revert objects.

CMVC Reports

- History Reports
  SHOW_HISTORY and SHOW_HISTORY_BY_GENERATION are used to look at what has changed. SHOW_HISTORY is used to look at changes between views and/or configurations.
  SHOW_HISTORY_BY_GENERATION is used to look at arbitrary sequences of changes

- Show_Out_Of_Date_Objects
  This report tells you what objects in the source view have been changed in some other view, tells you how many generations out of date you are, and where the last changes happened.

- Show
  Gives the above information on an object basis.

- Show_All_Controlled
  Gives the above imformation on all controlled objects in the view.

- Several Others - See the Cmvc Spec

Disk Space

- Space used by Development of Cmvc
  The following summarizes the space used by the Cmvc subsystem (the source for Cmvc). It was obtained by running !IMPLEMENTATION.CMVC_IMPLEMENTATION_UTILITIES.ANALYZE_SPACE The space numbers are in pages (1024 bytes). The database is 4 months old.

```
Cmvc_Database                          :    Today
11:21:31 Drk      3533k { 0}

Space Analysis For CMVC Database
!ENVIRONMENT.CMVC.STATE.CMVC_DATABASE
Space Usage

    CONFIGURATIONS => 19
    CONFIGURATION_PAGES => 1
    MEMBER_LIST_PAGES => 56
    ELEMENTS => 157
    UNUSED_ELEMENTS => 0
    ELEMENT_PAGES => 5
    RESERVATION_TOKENS => 301
    UNUSED_RESERVATION_TOKENS => 1
    RESERVATION_TOKEN_PAGES => 157
    GENERATIONS => 1034
    GENERATION_PAGES => 315
    TEXT_HEADERS => 1165
    TEXT_HEADER_PAGES => 36
    TEXT_PAGES => 2198
    RANGE_PAGES => 634
    STRING_PAGES => 25
    FREE => 8
    NAME_XLAT_MAPS => 15
    Unnaccounted_for_pages => 0
```

Topics Requiring Drawing Pictures

- Multi Target Development
  Issues relate to how reservation tokens are shared.

- Multi Release Development
  Issues relate to how changes are propagated.

- Merge_Changes
  Main issue is how to read the reports.

Code Views

- Make_Code_View

  — All units should be coded error if not need bodies

  — Main units will become loaded main units

- Non-Ada Objects

  — Copied as-is

- Imports

  — Named by Object Ids and string names. If object id becomes invalid, string name will be used for resolution from then on, slowing loading substantailly.

  no links, no imports
  model is there, but unused

- Information

  Cmvc_Maintenance.Display_Code_View

Compatibility and Incremental Coding

- Purpose
  - — Compatibility guarantees that each declaration that is different in a subsystem is allocated a different offset it its package, and that each declaration that is the "same" is allocated the same offset.
  - — Declarations are the "same" if they are syntactically identical.

Compatibility and Incremental Coding cont'd...

- Mechanism

  — During the install and coding process, each declaration in any library visible part is assigned a unique declaration signature and that signature is allocated an offset. This information is recorded in the Compatibility Database.

  — The offset assigned in independent of the order of declarations in a unit.

  — Thus, an incremental insertion will be runtime compatible regardless of where and when it is made.

  — When a declaration is deleted, its offset is not reused.

Garbage collecting offsets

- Fragmentation

  — As many changes are made in a spec in a subsystem, offsets will gradually become fragmented. This will eventually slow execution. offsets past 128: instructions one longer
  more pages required, less locality

- Cmvc_Maintenance.Destroy_Cdb

  — Demotes all units in a subsystem and deletes its compatibility database

  — When promoted, new offsets will be assigned to declarations

  — Any code view on other machines will be obsolesced and should be deleted. If they are left around, they could be radically incompatible with spec views.

  - will destroy code views on some machine

Displaying compatibility/offset information

- Cmvc_Maintenance.Display_Cdb
  - — Shows some information: unit numbers and number of declarations in each unit
  - — There is presently no way to display offsets
  - — You can write a unit that references what you are interested in and then look at the offsets that were generated in your code

put_(<declaration>'offset)
Integer'Image( )

Primaries and Secondaries

- Primary Subsystem

  — Development is done in the primary. New declarations can be added.

- Secondary Subsystem

  — This is an inactive copy. No new declarations can be added that are not already in the compatibility database.

  — You can use Archive.Copy with option CDB to update compatibility information in a secondary

  Cmvc_Maintenance.Update_Cdb -fetch from primary

  — When a unit is moved from a primary to a secondary via Archive, compatibility information is moved with it

  — If you want to make an incremental insertion in a secondary, the compatibility information for the new declaration must be moved first

Subsystem Id

- Each subsystem has a unique id number.
  - — This can be displayed using the CMVC_Maintenance.Display_Cdb command or by looking in the State file in the Compatibility world in the State directory in the subsystem root.
  - — The State file contains the subsystem id and the machine on which the primary was last known to reside.
- Archive will refuse to move information into a subsystem if the subsystem id does not match
- Cmvc_Maintenance.Make_Primary
  - — This used to change a secondary into a primary.
  - — Parameter controls whether this is a move, or a new subsystem.

Archive.Save/Restore of Subsystems

- Restoring makes secondary
  - — Option Primary - Causes restored subsystem to be a primary.
- Trailing_Blanks option
  - — Default is 2 - preserves already set line breaks
  - — Value 0 causes all line breaks to be considered significant. Can be used when converting from Gamma if people care about their line breaks.
- Saving/Restoring compatibility information
  - — Use the CDB option

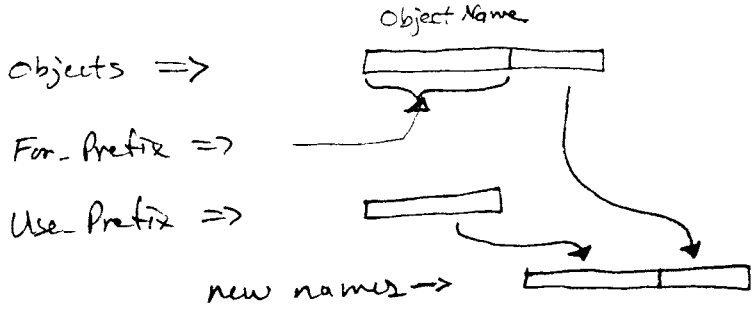
Archive.Save/Restore cont'd...

- Don't use renaming options when moving code

  — This can change the import references so they no longer connect to anything. This cannot be recovered without reloading the view.

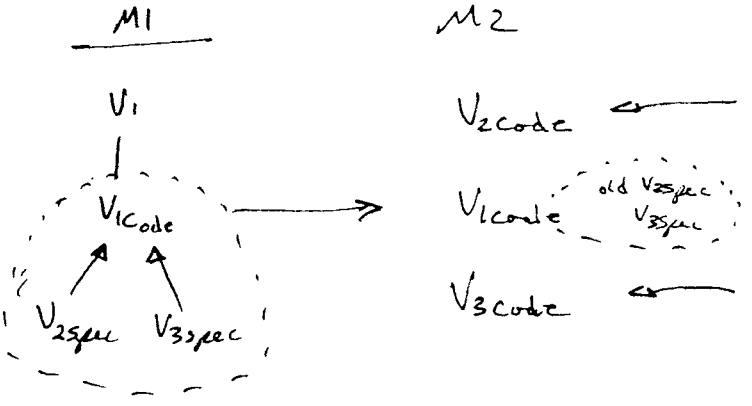
names buried inside the code to referenced objects
those names are renamed with the For/Use_Prefix parameters
could result in unintended name changes

Restore/Copy



Main Unit Issues

- Pragma Main Ignored in Spec Views
- Main units in code views
  - — They become loaded main units when the code view is built. This means that they are no longer affected by changing imports.
  - — This may be unexpected: A likely scenario is that the code view is moved to an integration machine to be combined with other subsystems. The main unit will work as it was when the code view was created on the development machine.



avoid pragma main in load views
put in unit outside of subsystem to get functionality

WORK ORDERS

- Capture History By Task
  Work Orders are intended to capture activity performed to accomplish a task, as opposed to capturing changes over a time to some object.

- All CMVC Activity Is Noted
  All CMVC operations are noted in the work order. This provides an audit trail of activity.

- Works With Source Management
  Work Orders, combined with Source Management, allow a user or project manager to see exactly what changed, and who changed it.

Ventures

"Project"

- Collections Of Work Orders
  A Venture is a collection of work orders. It defines the structure of the work order and specifies the defaults. The venture is especially useful for constructing reports.

- Contains Default Work Orders
  The venture specifies the default work orders for the people using the venture.

- Specifies Policies
  There are various policies that can be specified using the Venture. These vary from requiring comments during CMVC operations to requiring a default work order.

Work Order Lists

- Arbitrary Collections Of Work Orders
  A Work Order List is a collection of work orders from the same venture. This can be used as an organizational aid.

- Work Orders Can Be On Multiple Lists

- Work Orders Can Be On No List

What Is In A Work Order

- Status
  Pending, In_Progress, Closed

- Fields
  User (customer) definable information. These are provided so the user can customize work orders to address the problem at hand.

- Users
  A list of all users who have done any CMVC operations while attached to the work order.

- Configurations
  A list of configurations created/touched while the work order is in effect.

- Versions
  A list of all version/elements that have been touched while the work order is in effect.

- Comments
  A list of all comments supplied to CMVC operations while the work order is in effect. In addition, CMVC itself generates comments on occasion, and logs them here. Policy can be changed to force comments

- Notes
  Description - inserted when created

Operations

- Intended For Programmatic Use
  The programmatic interface is very rich. The command line interface is much weaker.

- Object Editors
  There are object editors for Ventures, Work Orders, and Work Order Lists. These are not fully functional; many items cannot be modified via the object editor interface.

- Command Line Interface
  Has enough gumption to create work orders, etc., set defaults, and simple enquiries.

Issues

- Copying Work Orders
  Lib.Copy will copy work orders, but doing so isn't useful. They are not registered in the Venture, and operations on them might fail. Use Archive.Copy to copy them.

- Policies
  If the policy Require_Comments is true, all CMVC operations must have comments. This has met with some resistance. Try the Require_Comments_At_Check_In if this is a problem.

  Don't forget about the Require_Default_Venture policy, available through the Work_Order_Implementation package.

- Functionality
  It is likely that programs will have to be written to solve real customer problems. It isn't clear who is going to do this. For example, much has been said about a problem reporting system. It isn't clear where this will come from.

Venture Content

```
!Environment.Cmvc.Work_Orders.Cmvc_Venture
Notes: ""

Policy_Switches:
  Require_Current_Work_Order  => True
  Require_Comment_At_Check_In => True
  Require_Comment_Lines       => False
  Journal_Comment_Lines       => True
  Allow_Edit_Of_Work_Orders   => False

Fields:

Work_Orders:
(!Environment.Cmvc.Work_Orders...)
  ...Build             : In_Progress;
  ...Cbh_Test          : In_Progress;
  ...Drk               : In_Progress;
  ...Relocation_Tests  : In_Progress;

Default_Work_Orders:
(!Environment.Cmvc.Work_Orders...)
  Cbh.S_1 => ...Cbh_Test

Work_Order_Lists:
(!Environment.Cmvc.Work_Orders...)
  ...Mtd

Default_Work_Order_Lists:
```

Work_Order_Content

```
!Environment.Cmvc.Work_Orders.Build : In_Progress;
Notes: "implementing build and accept from configuration"

Parent Venture: (!Environment.Cmvc.Work_Orders...)
  ...Cmvc_Venture

Status: In_Progress
  Created at 87/05/22 10:12:33 by Cbh.S_1

Fields:

Comments: 133
  87/06/04 14:51:05 Mtd.S_1 for "" => "Initial:hello"
  87/06/04 14:51:43 Mtd.S_1 for "State.Release_History" =>
    "COPY/RELEASE: Creating Release History object in new view"
  87/06/06 14:49:18 Mtd.S_1 for "Units.Cmvc.Cmvc.Destroy_View'Body" =>
    "CHECK_OUT: Bogus check out attempting to promote unit from archived to source"
  87/06/06 14:50:07 Mtd.S_1 for "Units.Cmvc.Cmvc.Destroy_View'Body" =>
    "CHECK_IN: Bogus check out attempting to promote unit from archived to source"

Users: 1
  Mtd.S_1

Versions: 2 (!Environment.Cmvc...)
  87/06/06 14:50:07 "Units.Cmvc.Cmvc.Destroy_View'Body".9
...Configurations.Rev9_Mtd_Working

Configurations: 0
```

```
with Compilation;
with System_Utilities;

package Cmvc is

    -- All CMVC commands raise Profile.Error if any error is detected
    -- and Profile.Propagate or Profile.Raise_Error is true

    ------------------------------------------------------------------------

    -- Some of the following reservation commands take the name of an object
    -- that appears in more than one view.  The naming expression
    --      !mumble.subsystem.[view1, view2, view3].units.object
    -- is useful for such times.

    procedure Check_Out (What_Object : String := "<CURSOR>";
                         Comments : String := "";
                         Allow_Demotion : Boolean := False;
                         Allow_Implicit_Accept_Changes : Boolean := True;
                         Expected_Check_In_Time : String := "<TOMORROW>";
                         Work_Order : String := "<DEFAULT>";
                         Response : String := "<PROFILE>");

    -- Check out reserves one or more objects (specified by What_Object) so
    -- that they may be modified in only one view.  All of the
    -- objects specified must belong to the same working view.
    -- An object must be 'controlled' to be reserved (see Make_Controlled),
    -- a warning is issued for objects that are not controlled.

    -- The reservation spans all of the views that share the
    -- same reservation token for the element.

    -- This command implicitly accepts changes in the checked out object,
    -- updating the value of the object to correspond to the most
    -- recent generation of that element/reservation token pair.

    -- The Comments field is stored with the notes for the object.
    -- If What_Object is a set, the comment is stored with all of them.

    -- Expected_Check_In accepts any string that Time_Utilities.Value
    -- will accept.

    procedure Check_In (What_Object : String := "<CURSOR>";
                        Comments : String := "";
                        Work_Order : String := "<DEFAULT>";
                        Response : String := "<PROFILE>");

    -- Release the reservation on the object.  What_Object may
    -- specify a set of objects.  This command only applies to
    -- the controlled objects in the set and will note any
    -- objects that are not controlled.

    -- Comments are treated as in Check_Out



    procedure Accept_Changes (Destination : String := "<CURSOR>";
                              Source : String := "<LATEST>";
                              Allow_Demotion : Boolean := False;
                              Comments : String := "";
```



```
                              Work_Order : String := "<DEFAULT>";
                              Response : String := "<PROFILE>");

    -- This operation updates the Destination to reflect changes
    -- (objects that have been checked in) specified by Source.

    -- The Destination is either a view or a set of objects (all in
    -- one view).  Specifying the view is equivalent to specifying
    -- all the objects in the view.  Uncontrolled objects in the
    -- destination are ingored except that a note is issued.

    -- The Source is either "<LATEST>", a view, a configuration,
    -- or a set of objects all in one view.

    -- If the Source is "<LATEST>", the destination objects
    -- will be updated to the most recently checked in version.
    -- If the most recent generation of a source object is currently
    -- checked out, the previous generation is used and a warning
    -- is issued.

    -- If the Source is a view and the Destination is a view, this command
    -- is basically "Make the Destination view look exactly like the
    -- Source view".  Every controlled object in the source is copied
    -- to the destination and the configuration in the destination
    -- is updated.  This includes new objects which did not previously
    -- exist in the destination.  If the destination has a more recent
    -- version than the source, the destination will not be updated and
    -- a warning is issued.  In particular, if objects are checked out in
    -- the destination, they will not be changed.
    -- If objects are checked out in the source this operation
    -- will use the previously checked in version of the object and
    -- a warning will be issued.

    -- If the Source is a view and the Destination is a set of objects,
    -- the destination objects are updated to the corresponding objects
    -- in the source view, as above.

    -- If the source is a configuration it is identical to having the
    -- source be a view except that the configuration specifies the
    -- versions to use and they may be older (less up to date) than
    -- the ones in the destination.  Thus if the source is a configuration
    -- then destination objects may "go backwards", while this will not
    -- happen if the source is a view.

    -- If the source is a set of objects and the destination is a view,
    -- the corresponding objects in the destination view are updated
    -- to the source objects.

    -- A common way of using  Accept_Changes is to use the default parameters
    -- during normal development to accept changes made in other subpaths.
    -- Then periodically an integration view (in the path) is updated by
    -- first accepting all relevant subpaths into the integration view
    -- (accept_changes (destination => integration_view, source =>
    -- active_subpath_working_view)).
    -- Then this integration view is compiled (and tested).  The subpaths are
    -- then re-synchronized by accepting the integration view (source =>
    -- integration_view, destination => destination_subpath_working_view).

    -- In addition to synchronizing the source, this protocol updates
    -- the libraries in such a way the relocation operates most effectively,
```

```
-- preventing compilation in many cases when changes move between views.

procedure Abandon_Reservation (What_Object : String := "<SELECTION>";
                               Comments : String := "";
                               Work_Order : String := "<DEFAULT>";
                               Response : String := "<PROFILE>");

-- Forget about a check_out of some object, or set of objects.
-- This reverts the objects back to last checked in version.
-- This operation is an "undo" for Check_Out, except that it
-- does not undo the implict Accept_Changes that goes with
-- a Check_Out.

procedure Revert (What_Object : String := "<SELECTION>";
                  To_Generation : Integer := -1;
                  Make_Latest_Generation : Boolean := False;
                  Comments : String := "";
                  Work_Order : String := "<DEFAULT";
                  Response : String := "<PROFILE>");

-- Replace the contents of the specified object with the contents
-- of the specified generation.  The operation is equivalent to an
-- Accept_Changes from a configuration containing the specified
-- generation.

-- If Make_Latest_Generation is true, then the operation is equivalent to
-- a Check_Out, a copy of the specified generation into the object, and
-- a Check_In.

-- Generation of -n means n generations back; thus -1 => the previous
-- generation.

------------------------------------------------------------------------


-- The following commands allow the creation and interogation of
-- a note scratchpad for each element.  Descriptive information
-- regarding what is being changed, why, or whatever, can be put
-- into the scratchpad.

procedure Get_Notes (To_File : String := "<WINDOW>";
                     What_Object : String := "<CURSOR>";
                     Response : String := "<PROFILE>");

-- Copy the notes from the object.  If To_File is the default, then
-- a new I/O window is created and the notes are copied into this window.
-- The first line of this window is the name of the object, which is
-- used by Put_ and Append_Notes to put the notes back.  The notes
-- displayed are those that go with the generation of the object pointed
-- at.  See Cmvc_History for ways of getting notes and other information
-- on a range of generations


-- The next three commands require the object in question to be
-- checked out.

procedure Put_Notes (From_File : String := "<WINDOW>";
                     What_Object : String := "<CURSOR>";
                     Response : String := "<PROFILE>");
```



```
-- Replace the notes for the specified object.  If the I/O window
-- was created by Get_Notes, the window (first line) contains the name
-- of the object to write back into, and What_Object is ignored.

procedure Append_Notes (Note : String := "<WINDOW>";
                        What_Object : String := "<CURSOR>";
                        Response : String := "<PROFILE>");

-- Append the specified text to the notes.  If Note is <IMAGE_TEXT>,
-- the associated window must have been created by Get_Notes or
-- Create_Empty_Note_Window; in this case What_Object is ignored.
-- If note is a string, then that string is appended to the object
-- selected by What_Object.  If the content of Note is prepended with a
-- '_', Note is interpreted as a text file name, and the content of
-- that file is appended to the selected object.

procedure Create_Empty_Note_Window (What_Object : String := "<CURSOR>";
                                    Response : String := "<PROFILE>");

-- Create an empty window (with no underlying directory object)
-- to be used for constructing notes for the specified object.
-- Typically, Append_Notes is used to actually add the text
-- to the object's notes.

------------------------------------------------------------------------


procedure Make_Controlled
             (What_Object : String := "<CURSOR>";
              Reservation_Token_Name : String := "<AUTO_GENERATE>";
              Join_With_View : String := "<NONE>";
              Comments : String := "";
              Work_Order : String := "<DEFAULT>";
              Response : String := "<PROFILE>");

-- Make the object or objects specified by What_Object be subject to
-- reservation.  The objects must be in a working view and not
-- already controlled.  All objects must be in the same subsystem.
-- If Join_With_View is specified, the objects are joined with the
-- object in that view, using the reservation token specified by that view.
-- If no view is specified, the reservation token name is used if provided,
-- else the development path name of the view containing the object is
-- used as the reservation token name.

procedure Make_Uncontrolled (What_Object : String := "<CURSOR>";
                             Comments : String := "";
                             Work_Order : String := "<DEFAULT>";
                             Response : String := "<PROFILE>");

-- Make an object or objects uncontrolled.
-- This means the objects are no longer subject to reservation
-- (in the enclosing view).

procedure Sever (What_Objects : String := "<SELECTION>";
                 New_Reservation_Token_Name : String := "<AUTO_GENERATE>";
                 Comments : String := "";
                 Work_Order : String := "<DEFAULT>";
                 Response : String := "<PROFILE>");

-- Make the object(s) in the given working view(s) have a separate
```

```
-- reservation.  This command severs the relationship between views
-- for objects.  When done, the views specified in this command will
-- have their own reservation to share.  All other views (not
-- specified) will share a different reservation.
-- A specific reservation token name can be provided, if desired.

procedure Join (What_Object : String := "<SELECTION>";
                To_Which_View : String := ">>VIEW NAME<<";
                Comments : String := "";
                Work_Order : String := "<DEFAULT>";
                Response : String := "<PROFILE>");

-- Make object in two or more working views share a reservation.  The
-- objects in the views must be identical (textually)  and controlled
-- for this command to succeed.

------------------------------------------------------------------------


procedure Merge_Changes (Destination_Object : String := "<SELECTION>";
                         Source_View : String := ">>VIEW_NAME<<";
                         Report_File : String := "";
                         Fail_If_Conflicts_Found : Boolean := False;
                         Comments : String := "";
                         Work_Order : String := "<DEFAULT>";
                         Response : String := "<PROFILE>");

-- Merge two versions of the same object together, leaving the result
-- in destination object.  In order for this command to succeed, the
-- Source_View and the view containing the Destination_Object must
-- have been copied from some common view sometime in the past, and
-- the configuration for that view must still exist.

-- Destination_Object must refer to the last generation; all changes must
-- have been accepted.

-- The command writes a report showing what it did, as well as changing
-- the destination object.  If the report_file name is "", the report
-- is written to Get_Simple_Name (Destination_Object) & "_Merging_Report".

-- Conflicts are defined to be regions of change in the source and
-- destination that directly overlap, ie the same line(s) have been
-- changed in both objects.  If Fail_If_Conflicts_Found is true,
-- no updating is done, but the report file is left.

-- If it is desired to rejoin the two objects after the merge, then
-- check out the Merge source object, copy the Merge Destination_Object
-- into the source, then Join the objects.

------------------------------------------------------------------------

function Imported_Views (Of_View : String := "<CURSOR>";
                         Include_Import_Closure : Boolean := False;
                         Include_Importer : Boolean := False;
                         Response : String := "<WARN>") return String;

-- return a string suitable for name resolution that names the union of
-- all of the imports specified by the view(s) Of_View.  These views
-- are in no particular order.
```



```
------------------------------------------------------------------------
--                              IMPORTS                               --
------------------------------------------------------------------------

-- CMVC supports selective importing of units when views are imported.
-- This is accomplished using Imports_Restrictions and
-- Exports_Restrictions.
--
-- Exports_Restrictions are subsets of exported Ada units controlled
-- by the exporting view (spec view).  The subset is determined by the
-- contents of a text file in the Exports directory of the view.  This
-- file contains Naming expressions which, when resolved against the
-- Units directory, produce a list of objects that are exported by
-- that subset.

-- Imports_Restrictions are further restrictions on what Ada units are
-- to be imported.  The restriction specifies which export restriction
-- to use (if any), a list of Ada units (using simple names) to
-- exclude, and a list of units to rename.  A restriction is a text
-- file, in the Imports directory, with the same name as the subsystem
-- containing the view being imported.  Each line of the file
-- specifies one thing.  The form of the lines are:
--
--   EXPORT_RESTRICTION=>restriction_name
--       Specify the name of the export restriction.  No blanks are
--       allowed.  If more than one restriction is specified, the
--       union of all of the restictions is used.
--   Object_Name      Link_Name
--       Import Object_Name but make a link with Link_Name (a rename)
--   ~Object_Name
--       Dont import Object_Name
--   Object_name
--       Import Object_Name and use Object_Name for the link name
--   @
--      Import all Objects, except those removed above
-- In all cases, the names provided above are simple names, ie no '.'s
-- in them.

------------------------------------------------------------------------
--                           SELECTING VIEWS                          --
------------------------------------------------------------------------

-- In the following commands, wherever a view is called for, a naming set
-- can be used.  A text file containing the names of configurations
-- or views can also be used.  However, you must use the leading '_'
-- convention supported by Naming.  Also, configuration names can be
-- used in place of views anywhere, assuming that the view represented
-- by the configuration still exists.


------------------------------------------------------------------------
--                             SPEC VIEWS                             --
------------------------------------------------------------------------

-- Spec views in CMVC are by default uncontrolled.  The reason for this
-- is to allow free changing of specs in the load views, accepting the
-- changes back and forth, then incrementally making the changes in the
-- spec views.

-- It controlling of spec views is desired, use Make_Controlled after
```

```
-- creating the views.  But be forewarned that checking out a spec
-- where an implicit accept is required will probably obsolesce all
-- of the spec's clients.

------------------------------------------------------------------------


procedure Release (From_Working_View : String := "<CURSOR>";
                   Release_Name : String := "<AUTO_GENERATE>";
                   Level : Natural := 0;
                   Views_To_Import : String := "<INHERIT_IMPORTS>";
                   Create_Configuration_Only : Boolean := False;
                   Compile_The_View : Boolean := True;
                   Goal : Compilation.Unit_State := Compilation.Coded;
                   Comments : String := "";
                   Work_Order : String := "<DEFAULT>";
                   Volume : Natural := 0;
                   Response : String := "<PROFILE>");

-- Create a new release view in the subsystem.  If Release_Name is
-- "<AUTO_GENERATE>", the view will have the same name prefix as the
-- working view, with _n_m appended as appropriate given the level.
-- Otherwise Release_Name must be the simeple name of the new release.

-- Since the new view is a release, it is frozen.  If From_Working_View
-- names multiple views, each named working view is released as
-- above, and the imports are adjusted so that the new releases
-- reference each other as appropriate instead of the working views.
-- Views_To_Import specifies, perhaps by indirection through an activity,
-- a set of views to be used as imports by the new view(s).  This allows
-- changing imports during a release.  Imports already adjusted during
-- the releasing of working views will be left alone, otherwise
-- subsystems currently imported will be reimported.  In other words,
-- if this were an import command, Only_Change_Imports would be true.

-- If Compile_The_View is true, the compiler is run before the views
-- are frozen, trying to promote the units to the indicated Goal.
-- The views are frozen even if compilation fails.

-- This command creates a configuration object named
-- SUBSYSTEM.state.configurations.release_name.  It also creates an
-- import description file in the same place, named release_name &
-- "_imports".  This import description file lists the configuration
-- objects for all views that are imported.  It is maintained by
-- all commands that modify or adjust the imports.  These two objects
-- are used to reconstruct views from configurations.

-- A controlled text object (state.release_history) is used by this
-- command.  Release enters the comments supplied with the command
-- into the notes for this object.  Feel free to check out and modify
-- this object to further describe what is going on.  This object is joined
-- across all of the releases and the working view of a subpath.
-- Furthermore, the object is checked out and in by the release command
-- in order to mark the time of the release.


------------------------------------------------------------------------


procedure Copy (From_View : String := "<CURSOR>";
```



```
                New_Working_View : String := ">>SUB/PATH NAME<<";
                View_To_Modify : String := "";
                View_To_Import : String := "<INHERIT_IMPORTS>";
                Only_Change_Imports : Boolean := True;
                Join_Views : Boolean := True;
                Reservation_Token_Name : String := "";
                Construct_Subpath_Name : Boolean := False;
                Create_Spec_View : Boolean := False;
                Level_For_Spec_View : Natural := 0;
                Model : String := "<INHERIT_MODEL>";
                Remake_Demoted_Units : Boolean := True;
                Goal : Compilation.Unit_State := Compilation.Coded;
                Comments : String := "";
                Work_Order : String := "<DEFAULT>";
                Volume : Natural := 0;
                Response : String := "<PROFILE>");

-- Create a new working view.  Working views are named Mumble_Working,
-- where mumble is supplied as New_Working_View.  If Join_Views is
-- true, the two views share reservations of the all of the controlled
-- objects in the two views.  If false, reservations aren't shared
-- across the views for any objects.  If From_View names multiple views, a
-- copy is made for each of those views and, if the originals
-- import each other (computed using the subsystem, not the view),
-- the copies will (try) to import the new views of those subsystems.

-- If Join_Views is false, new reservation tokens are created for all
-- of the controlled objects.  The default is to use the name supplied
-- as the >>SUBPATH_NAME<<.

-- View_To_Import supplies a set of views to be processed according to
-- the value of Only_Change_Imports.  If Only_Change_Imports is true,
-- a copied view always inherits the source view's imports.  After the
-- copy, the imports specified by View_To_Import are applied against the
-- new view, replacing any inherited import if needed.
-- If Only_Change_Imports is false, then either the imports are inherited
-- from the source, or the complete set of imports specified by
-- by View_To_Import is imported into the copy.

-- View_To_Modify specifies the set of working views that are to have
-- their imports changed to refer to the new copy(s).  The
-- View_To_Modify views are also changed to refer to the views specified
-- by View_To_Import.  For this import operation, Only_Change_Imports
-- is forced to true.

-- Construct_Subpath_Name cause Copy to contruct the target view name
-- by appending New_Working_View to the prefix of the source view name
-- up to the first '_' (See paths and subpaths below).

-- Remake demoted units, if true, indicates that ada units that were
-- demoted during the copy process are to be recompiled.  They are
-- compiled to the level indicated by Goal.  Units are not complied
-- to a state higher that they were in the source.

-- Goal further indicates the desired state of all of the units after
-- copy.  No unit will be in a state higher than specified by goal, but
-- might be in a lower state.  For example, a source unit that is copied
-- will remain source, regardless of Goal, but a Coded unit will be
-- demoted if Goal is installed or less.
```

```
-- The order of the copy and import operations is:
--
--      1.  Create the new view.
--      2.  If Inherit_Imports, bring along the old imports
--      3.  Import the new views into the new views, forcing
--          Only_Change_Imports => True
--      4.  If not Inherit_Imports, import the specified views
--          into the new views.
--      5.  Import the new views + View_To_Import into Views_To_Modify,
--          forcing Only_Change_Imports => true

-- Spec views are created by copying the units if the source is a load
-- view, otherwise using Relocation.  Spec views are created with all
-- objects uncontrolled.  If level_for_spec_view = natural'last, the
-- spec view is given the name supplied as new_working_view, otherwise
-- a name is generated as 'New_Working_View & Release_Numbers & "_spec"'

-- It is recognized that this is a complicated command.  Using the
-- procedures below (which are effectively renames) might make more
-- sense if the methodolody in use permits it (Path, Subpath, etc).


------------------------------------------------------------------------
--                        PATHS AND SUBPATHS                          --
------------------------------------------------------------------------

-- The following procedures support the notion of paths and subpaths.
-- A Path is a logically connected series of releases in which all
-- controlled objects are joined together.  In other words, there is
-- no branching within a path.  A Subpath is an extension of the
-- path, allowing multiple developers to make changes and test
-- without getting in each others way.  However, controlled objects
-- in the subpaths are joined with the path; people in two subpaths
-- cannot independently change the same object. In addition, a path
-- and its subpaths share the same model, which means they share
-- the same Target_Key and initial links.

-- In Delta, paths and subpaths are identified by string name conventions.
-- The name of the path is the view name up to the first '_'.  The
-- subpath extension is the name from this '_' to the '_Working'. Thus
-- Rev9_Cbh_Working has a path name of Rev9 and subpath extension of
-- Cbh.

-- Multiple paths are used when multiple targets are involved, or when
-- objects are to be changed independently.  For example, assume that
-- a version of a product has been shipped, and is in maintenance, and
-- that development is progessing on a new version.  It is likely that
-- the old and new versions would be separate paths, since the objects
-- would have to be independently changed (these paths would not be
-- 'joined').

-- In the multiple target case, the paths might be created joined.
-- Using the above scenario, assume that the release that has been shipped
-- works on two targets, but most or all of the code is target
-- independent.  Then the two paths, one for each target, would be
-- created joined together, then have the objects that are not common
-- 'Sever'ed.

procedure Make_Path (From_Path : String := "<CURSOR>";
                     New_Path_Name : String := ">>PATH NAME<<";
                     View_To_Modify : String := "";
                     View_To_Import : String := "<INHERIT_IMPORTS>";
                     Only_Change_Imports : Boolean := True;
                     Model : String := "<INHERIT_MODEL>";
                     Join_Paths : Boolean := True;
                     Remake_Demoted_Units : Boolean := True;
                     Goal : Compilation.Unit_State := Compilation.Coded;
                     Comments : String := "";
                     Work_Order : String := "<DEFAULT>";
                     Volume : Natural := 0;
                     Response : String := "<PROFILE>");


procedure Make_Subpath (From_Path : String := "<CURSOR>";
                        New_Subpath_Extension : String := ">>SUBPATH<<";
                        View_To_Modify : String := "";
                        View_To_Import : String := "<INHERIT_IMPORTS>";
                        Only_Change_Imports : Boolean := True;
                        Remake_Demoted_Units : Boolean := True;
                        Goal : Compilation.Unit_State := Compilation.Coded;
                        Comments : String := "";
                        Work_Order : String := "<DEFAULT>";
                        Volume : Natural := 0;
                        Response : String := "<PROFILE>");

-- The Subpath_Extension is appended to the path name of the source
-- view (From_Path).  From_Path can actually name the path or any
-- subpath of the path.  The '_' between the path and subpath extension
-- is automatically provided.

procedure Make_Spec_View
            (From_Path : String := "<CURSOR>";
             Spec_View_Prefix : String := ">>PREFIX<<";
             Level : Natural := 0;
             View_To_Modify : String := "";
             View_To_Import : String := "<INHERIT_IMPORTS>";
             Only_Change_Imports : Boolean := True;
             Remake_Demoted_Units : Boolean := True;
             Goal : Compilation.Unit_State := Compilation.Coded;
             Comments : String := "";
             Work_Order : String := "<DEFAULT>";
             Volume : Natural := 0;
             Response : String := "<PROFILE>");

-- Make a spec view for a path.  Spec_View_Prefix is the string that
-- replaces the path and subpath name.  For example, if creating a
-- spec view from a subpath named rev9_cbh_working, with
-- Spec_View_Prefix => Env9, the result will be Env9_n_Spec, assuming
-- level => 0 and two levels are specified by the model.  N is a
-- number automatically generated from the current release number for
-- the path/subpath.  If level = natural'last, the name supplied as
-- Spec_View_Prefix is used for the name of the view, with no suffixes

------------------------------------------------------------------------


procedure Import (View_To_Import : String := "<REGION>";
                  Into_View : String := "<CURSOR>";
                  Only_Change_Imports : Boolean := False;
```

```
                    Import_Closure : Boolean := False;
                    Remake_Demoted_Units : Boolean := True;
                    Goal : Compilation.Unit_State := Compilation.Coded;
                    Comments : String := "";
                    Work_Order : String := "<DEFAULT>";
                    Response : String := "<PROFILE>");

   -- Imports spec or combined views as appropriate into the specified
   -- view(s).  The import specification can be a set of view names,
   -- in which case all views are imported, unless only_change_imports is
   -- true.  In this case only subsystems that were imported sometime in
   -- the past are reimported.  All others are ignored.

   -- The import description file mentioned in the release command is
   -- brought up to date by this command.

   -- If View_To_Import is "", then the imports of Into_View are refreshed.
   -- This means the various imported views are examined, and any new
   -- Ada specs are imported in to the current view.

   -- It is useful to invoke Import with Views_To_Import = Into_View and
   -- Only_Change_Imports is true.  This will cause a set of views to be
   -- changed to import each other.

   procedure Remove_Import (View : String := ">>VIEW NAME<<";
                            From_View : String := "<CURSOR>";
                            Comments : String := "";
                            Work_Order : String := "<DEFAULT>";
                            Response : String := "<PROFILE>");

   -- remove references to a previously imported view.

   procedure Remove_Unused_Imports (From_View : String := "<CURSOR>";
                                    Comments : String := "";
                                    Work_Order : String := "<DEFAULT>";
                                    Response : String := "<PROFILE>");

   -- Search through all of the Ada units in the view and examine the
   -- withs.  If no units in some imported view are referenced, remove
   -- that import.

   -- This command generates warnings if units in spec or combined
   -- views are referenced, but the view isn't imported.  Errors are
   -- generated if units in load views are referenced.

   procedure Replace_Model (New_Model : String := ">>NEW MODEL NAME<<";
                            In_View : String := "<CURSOR>";
                            Comments : String := "";
                            Work_Order : String := "<DEFAULT>";
                            Response : String := "<PROFILE>");

   -- Replace the model with the new one.  All units must be source.
   -- This command gets the switch file from the new model (if one
   -- was provided), readjusts the maximum levels (which affects future
   -- releases), and rebuilds the links.

   ------------------------------------------------------------------------


   type Subsystem_Type_Enum is (Spec_Load, Combined);
```



```
   procedure Initial (Subsystem : String := ">>SUBSYSTEM NAME<<";
                      Working_View_Base_Name : String := "Rev1";
                      Subsystem_Type : Subsystem_Type_Enum := Cmvc.Spec_Load;
                      View_To_Import : String := "";
                      Model : String := "R1000";
                      Comments : String := "";
                      Work_Order : String := "<DEFAULT>";
                      Volume : Natural := 0;
                      Response : String := "<PROFILE>");

   -- Build a new subsystem of the specified type.  Also create a working
   -- view and import as specified.  This command can be used to create
   -- an empty view in an existing subsystem.

   ------------------------------------------------------------------------

   procedure Information (For_View : String := "<CURSOR>";
                          Show_Model : Boolean := True;
                          Show_Whether_Frozen : Boolean := True;
                          Show_View_Kind : Boolean := True;
                          Show_Creation_Time : Boolean := True;
                          Show_Imports : Boolean := True;
                          Show_Referencers : Boolean := True;
                          Show_Unit_Summary : Boolean := True;
                          Show_Controlled_Objects : Boolean := False;
                          Show_Last_Release_Numbers : Boolean := False;
                          Show_Path_Name : Boolean := False;
                          Show_Subpath_Name : Boolean := False;
                          Show_Switches : Boolean := False;
                          Show_Exported_Units : Boolean := False;
                          Response : String := "<PROFILE>");


   -- Show various things about a view.  Please see Cmvc_History for
   -- ways of extracting other information about the controlled objects
   -- in the view.

   ------------------------------------------------------------------------


   procedure Destroy_View (What_View : String := "<SELECTION>";
                           Demote_Clients : Boolean := False;
                           Destroy_Configuration_Also : Boolean := False;
                           Comments : String := "";
                           Work_Order : String := "<DEFAULT>";
                           Response : String := "<PROFILE>");

   -- Destroy a view.  If Demote_Clients is false, the view can have no
   -- referencing views (clients); if it does, the destroy fails.  If
   -- Demote_Clients is true, the view is "remove_import"ed from those
   -- clients (which might cause lots of obsolescence), then the view is
   -- destroyed.  The configuration object for the view is left behind
   -- in its normal place (see Release, above) so the view can be
   -- reconstructed using "Build"

   procedure Destroy_Subsystem (What_Subsystem : String := "<SELECTION>";
                                Comments : String := "";
                                Work_Order : String := "<DEFAULT>";
                                Response : String := "<PROFILE>");
```

```
-- Destroy a subsystem.  There must be no views in the subsystem

-------------------------------------------------------------------------

procedure Build (Configuration : String := ">>CONFIGURATION NAME<<";
                 View_To_Import : String := "";
                 Model : String := "R1000";
                 Goal : Compilation.Unit_State := Compilation.Installed;
                 Limit : String := "<WORLDS>";
                 Comments : String := "";
                 Work_Order : String := "<DEFAULT>";
                 Volume : Natural := 0;
                 Response : String := "<PROFILE>");

-- Rebuild a view from history.  If Configuration_Object_Name refers to
-- a text file, that file is assumed to contain a list of configuration
-- object names to be built.

-- If View_To_Import = "", and if a text file exists with the name "same
-- as configuration_object" & "_imports", that text file is opened
-- after the views are built and imports are constructed from the views
-- or configuration objects named in that file.  Please note that copy,
-- initial, import, and remove_import will create and maintain such a
-- text file, so it is probably there.

-------------------------------------------------------------------------
--                        HISTORY COMMANDS                             --
-------------------------------------------------------------------------

-- The following commands display history information, in various
-- formats, of Cmvc controlled objects

procedure Show_History (For_Objects : String := "<CURSOR>";
                        Display_Change_Regions : Boolean := True;
                        Starting_Generation : String := "<CURSOR>";
                        Ending_Generation : String := "";
                        Response : String := "<PROFILE>");

-- Display the history for the specified objects.  If a view is
-- specified, all of the controlled objects in that view are displayed.
-- This history includes notes, checked_out and _in information, and
-- optionally the actual changes

-- If display_change_regions is true, the differences between a
-- generation and the previous one (n-1, n) are displayed.  The display
-- is in the form of regions where changes occurred similar to that
-- produced by File_Utilities.Difference(Compressed_Output=>True)

-- The first generation to display is determined by looking up
-- the object in the view(s) specified by Starting_Generation.  If
-- Starting_Generation = "", the display starts at generation 1.

-- The last generation to display is determined by Ending_Generation.
-- If E.._G.. is "", the last displayed is the latest one.  If E.._G..
-- is the name of a view, the generation specified by that view is
-- used as the last.

procedure Show_History_By_Generation
            (For_Objects : String := "<CURSOR>";
             Display_Change_Regions : Boolean := True;
```



```
             Starting_Generation : Natural := 1;
             Ending_Generation : Natural := Natural'Last;
             Response : String := "<PROFILE>");

-- In this case, All_Units means all of the units in the current
-- view.  Naming a view means all units in that view.

procedure Show_All_Uncontrolled (Object_Or_View : String := "<CURSOR>";
                                 Response : String := "<PROFILE>");

-- List objects that are not controlled.  Produces output only if an
-- object listed (or one in the units directory if a view is supplied)
-- is not under CMVC control

procedure Show_Image_Of_Generation (Object : String := "<CURSOR>";
                                    Generation : Integer := -1;
                                    Output_Goes_To : String := "<WINDOW>";
                                    Response : String := "<PROFILE>");

-- Reconstruct an image of some generation of the specified object.
-- The default (-1) indicates back up one generation from that of
-- Object.  Negative numbers are relative to the generation of Object,
-- positive numbers are actual generation numbers.
-- The result is written to current output unless a file name is
-- supplied in Output_Goes_To.

-------------------------------------------------------------------------

-- The following commands produce a report showing objects that
-- meet some criteria.  This report shows the following information
-- about each object.

-- Object Name  Generation  Where  Chkd Out  By Who  Expected Check In
-- ===========  ==========  =====  ========  ======  ==================
-- UNITS.FOO      5 of 8    VIEW     Yes      MTD    Apr 7, 1987

-- Object name is the element name (the name from the view down)

-- Generation is a pair.  The first number is the generation of
-- the object used to lookup the element.  The second number is
-- the highest generation produced.

-- Where is either the view containing a copy of the last generation
-- if the object is not checked out, or the view in which the object
-- is checked out.  In the case where the object is not checked out,
-- it is possible that there is no representative object, in which
-- case this field is blank.

-- Chkd Out is 'Checked Out'.  If this is yes, 'By Who' and
-- 'Expected Check In' provide more information.

-------------------------------------------------------------------------

procedure Show (Objects : String := "<CURSOR>";
                Response : String := "<PROFILE>");

-- Produce the information desribed above for the listed objects.
-- Also produces a report for each object showing which views
-- contain elements sharing a reservation token with the object.
```

```
    procedure Show_All_Checked_Out (In_View : String := "<CURSOR>";
                                    Response : String := "<PROFILE>");

    -- Look through all of the controlled objects in the supplied view, and
    -- display information about them if they are checked out anywhere

    procedure Show_Checked_Out_In_View (In_View : String := "<CURSOR>";
                                        Response : String := "<PROFILE>");

    -- Display information about all of the objects checked out in the
    -- view pointed at (or in)

    procedure Show_Checked_Out_By_User
                 (In_View : String := "<CURSOR>";
                  Who : String := System_Utilities.User_Name;
                  Response : String := "<PROFILE>");

    -- Display information about any object in the view that is checked out
    -- be the user given.  This command will find the object even if it is
    -- checked out in some other view, as long as it is controlled in the
    -- view referred to.

    procedure Show_Out_Of_Date_Objects (In_View : String := "<CURSOR>";
                                        Response : String := "<PROFILE>");

    -- Display information about all objects in the view that are not
    -- at the latest revision.

    procedure Show_All_Controlled (In_View : String := "<CURSOR>";
                                   Response : String := "<PROFILE>");

    -- Display information about all controlled objects in this view


    ------------------------------------------------------------------------
    --                      ARCHIVE COMMANDS                              --
    ------------------------------------------------------------------------

    procedure Make_Code_View (From_View : String := "<CURSOR>";
                              To_View : String := "";
                              Comments : String := "";
                              Work_Order : String := "<DEFAULT>";
                              Volume : Natural := 0;
                              Response : String := "<PROFILE>");

    -- Make a code view with the given name.  From_View must only
    -- name load and/or combined views.  If a load view is provided, no
    -- specs are copied; all specs are copied for combined views.
    -- This operation fails if any unit isn't coded, or any spec exists
    -- for which a body is required and one doesn't exist.



    pragma Subsystem (Cmvc);
    pragma Module_Name (4, 3704);
end Cmvc;
```

```
package Cmvc_Maintenance is
    procedure Expunge_Database (In_Subsystem : String := "<CURSOR>";
                               Response : String := "<PROFILE>");

    -- Free up space in the Database by first finding all configurations
    -- in the database that no longer have objects and destroying them,
    -- then destroying all elements and join sets (with all of their
    -- generations) that are no longer referenced.

    procedure Delete_Unreferenced_Leading_Generations
                (In_Subsystem : String := "<CURSOR>";
                 Response : String := "<PROFILE>");

    -- Not yet implemented

    procedure Convert_Old_Subsystem (Which : String := "<SELECTION>";
                                     Response : String := "<PROFILE>");

    -- Convert all of the views in a subsystem to CMVC subsystems.  This
    -- command can convert more than one subsystem per call.


    procedure Check_Consistency (Views : String := "<CURSOR>";
                                 Response : String := "<PROFILE>");

    -- Verify that all of the views are consistent with the CMVC invariants.
    -- Checks that:
    --      The configurations all exist and are correct.
    --      There are no dangling controlled objects.
    --      The imports are ok, and that all of the imported subsystems
    --          record the reference.
    --      Various other things.

    ------------------------------------------------------------------------
    -- User level commands for manipulating the compatibility database (CDB)
    -- associated with subsystems.
    ------------------------------------------------------------------------

    procedure Display_Cdb (Subsystem : String := "<CURSOR>";
                           Show_Units : Boolean := False;
                           Response : String := "<PROFILE>");

    -- Displays a summary of the information in the CDB. If "show_units"
    -- is true, then a summary of information for the units currently
    -- known in the subsystem is also displayed.


    procedure Make_Primary (Subsystem : String := "<SELECTION>";
                            Moving_Primary : Boolean := False;
                            Response : String := "<PROFILE>");

    -- Makes the subsystem into a primary subsystem with its own read/write
    -- CDB.  If the subsystem was a primary this operation is a no-op.  If
    -- the subsystem is a secondary then a new subsystem_id is assigned.
    -- If "moving_primary" is set to true, then the location of the
    -- primary for this subsystem is being moved and the current subsystem_id
    -- will be used.  When moving a primary the user must make sure
    -- that the original primary is either destroyed or converted into
    -- a secondary to prevent corruption of the CDB.
```



```
    procedure Make_Secondary (Subsystem : String := "<SELECTION>";
                              Response : String := "<PROFILE>");

    -- Makes the subsystem into a secondary with the same subsystem_id.


    procedure Destroy_Cdb (Subsystem : String := "<SELECTION>";
                           Limit : String := "<WORLDS>";
                           Effort_Only : Boolean := True;
                           Response : String := "<PROFILE>");

    -- Destroys the CDB  and all remnants of it in compiled units.
    -- This includes demoting ALL units in the subsystem to source
    -- and deleting all code-only views.  If "effort-only" is set
    -- to true, then the effects of the operation are computed
    -- and displayed.


    procedure Update_Cdb (From_Subsystem : String := "<ASSOCIATED_PRIMARY>";
                          To_Subsystem : String := "<SELECTION>";
                          Response : String := "<PROFILE>");

    -- Moves the CDB from one subsystem to another using the network
    -- if necessary.  Both subsystems must have the same subsystem_id.


    procedure Repair_Cdb (Subsystem : String := "<SELECTION>";
                          Verify_Only : Boolean := True;
                          Delete_Current : Boolean := False;
                          Response : String := "<PROFILE>");

    -- Will rebuild the CDB to be consistent with the currently compiled
    -- units in the subsystem.  If "verify_only" is true then the CDB
    -- will not be changed, but will be checked for consistency with
    -- the currently compiled units.  If "verify_only" is false and
    -- "delete_current" is true then the current CDB will be deleted
    -- and then rebuilt.  If the "verify_only" is false and
    -- "delete_current" is false then existing entries in the CDB
    -- will be verified and missing entries will be added.

    pragma Subsystem (Cmvc);
    pragma Module_Name (4, 3707);
end Cmvc_Maintenance;
```

```
with Machine;
package Archive is

    procedure Save (Objects : String := "<IMAGE>";
                    Options : String := "R1000";
                    Device : String := "MACHINE.DEVICES.TAPE_0";
                    Response : String := "<PROFILE>");

    -- Save a set of objects (files, Ada units, etc.) to a tape or directory
    -- such that they may be restored to their original form at a later time
    -- or on another system.

    -- The Objects parameter specifies the primary objects to be saved.  It
    -- can be any naming expression.  By default, the current image is saved
    -- unless there is a selection on that image, in which case the selected
    -- object is saved.  Normally, the specified object(s) and all contained
    -- objects are archived; this feature can be disabled.

    -- The Options parameter specifies the type of tape to be written and
    -- options to control what is saved.  The Options parameter for each of
    -- the Archive operations is written as a sequence of option
    -- names separated by spaces or commas.  Options with arguments are
    -- given as an option name followed by an equal sign followed by a
    -- value.

    -- The save options are:
    --
    --   FORMAT = R1000 | R1000_LONG | ANSI
    --      R1000
    --        Writes an ANSI tape with the data file followed by the index
    --        file.  The images of the objects being saved are written
    --        directly to the tape. This is the default.

    --      R1000_LONG
    --        like R1000 format but the data file is written to one ANSI tape
    --        and the index file to a second ANSI tape.

    --      ANSI
    --        Writes the data to a temporary file and then writes both index
    --        and data file to a tape using ANSI tape facilities.

    --   LABEL=(<any balanced string>)
    --      An identifying string written at the head of the archived data.
    --      The label parameter allows the user to specify a string that
    --      will be put at the front of the index file.  When a restore is
    --      done the label specified to the restore procedure will be
    --      checked against the one on the save tape.

    --   NONRECURSIVE
    --      Save only the objects resolved to by the Objects parameter.  Do
    --      not recursively save objects that are inside of other objects.
    --      The default is to save the objects mentioned in the Objects
    --      parameter and all objects contained in them.

    --      To save a world and a subset of its contents one can say:

    --      Save (Objects => "[!HJL?,~!HJL.ABC?,~!HJL.DEF?]", ...,
    --            Options => "R1000 NONRECURSIVE");

    --   AFTER=<time_expression>
    --      Only objects changed after the time represented by
    --      <time_expression> will be archived.  The <time_expression>
    --      should be acceptable to the time_utilities.value function.

    --   COMPATIBILITY_DATABASE (CDB) [=<Subsystems>]
    --      Causes the full compatibility database for each subsystem
    --      specified to be archived. If no subsystems are specified with
    --      the option, the Objects parameter specification is used instead.
    --      The NONRECURSIVE option does not affect the interpretation of the
    --      CDB specificaton even when it is obtained from the Objects
    --      parameter.

    --      When Ada units in a subsystem are archived, the relevant
    --      portions of the subsystem Compatibility Database is
    --      automatically archived with them.  Therefore, this option is
    --      required only in special situations, primarily when one needs to
    --      "sync up" a primary and a secondary subsystem.

    --      To archive just Compatibility Databases, use

    --      Save ("Subsystems", "CDB");

    --      To archive compatibility databases with other objects, use

    --      Save ("Other Stuff", "CDB=Subsystems");

    --      The "Subsystems" and "Other Stuff" specifications will usually
    --      describe disjoint sets of objects.

    --   PREFIX=<naming pattern>
    --      A naming pattern that is saved with the archived objects, which
    --      can be recalled as the For_Prefix when the data is Restored.
    --      When set to an appropriate value, the restorer need not know
    --      exactly the names of the archived objects to be able to restore
    --      them to a new place.  If this option is not given, the value
    --      used is derived from the Objects parameter and CDB
    --      option (if present) by expanding context-sensitive characters
    --      (such as ^ and $), expanding indirect file references, and
    --      removing all attributes.

    -- For downward compatibility the following options are provided.
    --
    --   GAMMA0
    --      write a tape which can be read on a Gamma0 system.
    --
    --   GAMMA1
    --      write a tape which can be read on a Gamma1 system.
    --
    --   VERSION=<archive_version_number>
    --      write a tape that can be read by a version of source
    --      earlier than the current one. The argument is a three digit
    --      integer. For example, version=210.


    -- The Device parameter can be set to the name of a directory.  In this
    -- case the index and data files are written to that directory.  The
    -- tape format option is irrelevant in this case.

    ------------------------------------------------------------------------
```

```
procedure Restore (Objects : String := "?";
                   Use_Prefix : String := "*";
                   For_Prefix : String := "*";
                   Options : String := "R1000";
                   Device : String := "MACHINE.DEVICES.TAPE_0";
                   Response : String := "<PROFILE>");

-- Restore an object or a set of objects from an Archive Tape.

-- If the archive is on a tape then the tape format option given to
-- Restore should be the same as that given during the save.  If the
-- archive is in a directory then the device parameter on the restore
-- should be set to that directory.

-- The Objects parameter may be any wildcard pattern specifying the
-- objects to be restored.
--
-- For example:
--     !USERS.HJL.CLI.TEST
--     [!USERS.HJL.TESTS.@, !USERS.HJL.LOGS.ABC]

-- The pattern in the Objects parameter is compared against the full
-- names of the saved objects. The objects whose names match the Objects
-- parameter specification are restored. If the name denotes an Ada
-- unit all of its parts are restored from the tape. If the name denotes
-- a world or directory all of its subcomponents are restored.

-- The Use_Prefix and For_Prefix parameters provide a simple means for
-- changing the names of the archived objects when they are restored.

-- If the Use_Prefix is the special default value, "*", the For_Prefix
-- is ignored and the objects are restored using the names they had when
-- they were saved.

-- If the Use_Prefix is not "*", it must specify the name of an object
-- into which the archived objects can be restored.  The name for a
-- restored object is derived from the name of the archived object by
-- replacing the shortest portion of the name matched by the For_Prefix
-- with the value of the Use_Prefix.  If the For_Prefix is "*" the
-- archived objects are restored using the Default_Prefix stored with
-- the archived data.

-- For example:
--
--   Restore (Objects    => "!A.B.C.D.E",
--            Use_Prefix => "!X.Y",
--            For_Prefix => "!A.B.C");
--
-- will restore to !X.Y.D.E.

-- If the name of the archived object does not have the For_Prefix as a
-- prefix, it is restored under its original name.

-- The For_Prefix may contain wildcard characters (#, @, ?) and the
-- Use_Prefix parameter may contain substitution characters (@ or #
-- only). (Not implemented in D0)

-- For example:
--
--   Restore (Objects    => "[!A.B.TEST1, !D.E.F.TEST2]"
--            For_Prefix => "?.@"
--            Use_Prefix => "!C.D.@");
--
--   will restore to !C.D.TEST1 and !C.D.TEST2

-- If the object named by the prefix of the target name of an object
-- being restored doesn't exist, that object will be created as a set of
-- nested worlds. So, for example, if the For_Prefix is !A.B and the unit
-- being restored is then !A.B.X.Y.Z and ...X.Y hasn't been saved on
-- the tape then !A, !A.B, !A.B.X, !A.B.X.Y will be created as worlds.
--
-- The following options are allowed in the Options parameter:
--
--   FORMAT and LABEL: options as in the save option.

--   COMPATIBILITY_DATABASE, (CDB) [=<Subsystems>]
--      Specifies that the Compatibility Databases for just the named
--      subsystems are to be restored.

--   NONRECURSIVE
--      prevents subcomponents of libraries and Ada units from being
--      implicitly restored. for example:

--       Restore
--          (Objects => "[!USERS.HJL, !USERS.HJL.CLI, !USERS.HJL.CLI.@]",
--           Options => "R1000 NONRECURSIVE");

--       will restore only the named objects and not their substructure.

--   OVERWRITE = ALL_OBJECTS | NEW_OBJECTS | UPDATED_OBJECTS | CHANGED_OBJEC\
TS
--       ALL_OBJECTS
--          All specified objects are restored.  This is the default.

--       NEW_OBJECTS
--          Only specified objects that don't already exist on the target
--          machine are restored.

--       UPDATED_OBJECTS
--          Only specified objects that already exists on the target are
--          restored, but only if the update time of the archived object
--          is greater than the update time on the target object.

--       CHANGED_OBJECTS
--          Restore both new and updated Objects.

--   PROMOTE
--      After they are restored, any Ada units will be promoted to the
--      state they were in when they were archived.

--   REPLACE
--      Given an object that is being restored that already exists
--      on the target, this option will cause the restore operation

--        (1) to unfreeze the target object if it is frozen.

--        (2) If the target object is an installed or coded Ada unit
--            with clients, it is demoted to source using Compilation.
--            Demote with the "<ALL_WORLDS>" parameter.
```

```
--        (3) if the parent library into which an object is being
--            restored is frozen, the parent will be unfrozen to restore
--            the object then refrozen.
--
--    OBJECT_ACL=<acl_value>
--    WORLD_ACL=<acl_value>
--    DEFAULT_ACL=<acl_value>
--       Specifies the Access Control List for restored objects
--       (OBJECT_ACL) and worlds (WORLD_ACL) and the default ACL for
--       restored worlds (DEFAULT_ACL).  The value is either an ACL
--       specification or the special values INHERIT or ARCHIVED.
--       ARCHIVED means to use the ACL archived with the object and is
--       the default for all three ACL options.  INHERIT means to use the
--       standard inheritence rules for new versions of objects.

--    BECOME_OWNER
--       Modify the ACL of all restored objects such that the restorer
--       becomes the owner of the restored object.

-------------------------------------------------------------------------------

procedure List (Objects : String := "?";
                Options : String := "R1000";
                Device : String := "MACHINE.DEVICES.TAPE_0";
                Response : String := "<PROFILE>");

-- Produce a listing of the names of the objects on an Archive tape.

-- The Objects parameter specifies the objects to be listed.  Wildcards
-- are permitted, so if Objects = "?", the default, then all Objects are
-- listed.

-- The Options parameters are:

--    FORMAT and LABEL
--       as in the Save options.

-------------------------------------------------------------------------------

procedure Copy (Objects : String := "<IMAGE>";
                Use_Prefix : String := "*";
                For_Prefix : String := "*";
                Options : String := "";
                Response : String := "<PROFILE>");

-- Copy objects from one location to another, including between
-- machines on the same network.

-- The Objects parameter specifies where the objects are to be gotten
-- from as in an Archive.Save. The Use_Prefix/For_Prefix parameters
-- specify where the objects are to go as in Archive.Restore.

-- Each name consists of an (optional) machine name followed directly
-- by a Objects parameter. A machine name has the form !!name.
-- the Objects part of the source name is like that given to the save
-- operation.
--
-- The Use_Prefix and the For_Prefix function as in the Restore command.

-- If the Use_Prefix parameter is "*" or just a machine name, then the
```



```
-- source Objects are moved to the same place on the destination machine
-- as specified by the source.  The For_Prefix parameter is ignored.

-- If neither Objects nor Use_Prefix have a machine name then the
-- objects are copied from the source to the Use_Prefix on the
-- current machine.
--
-- The Options parameter has the following options.
--
--   AFTER=<time_expression>
--      as in the save operation.
--
--   COMPATIBILITY_DATABASE, CDB
--   NONRECURSIVE
--      as in the save operation.
--
--   PROMOTE, REPLACE,
--   BECOME_OWNER,
--   OBJECT_ACL, WORLD_ACL, DEFAULT_ACL
--      as in the restore operation.
--

-- Examples of calls:
--
--  Copy (Objects    => "!USERS.HJL.CLI",
--        Use_Prefix => "!!M1");
--
--    will copy the CLI directory in !USERS.HJL on the
--    current machine to machine M1 !USERS.HJL.CLI.
--
--  Copy (Objects => "!!M2!USERS.JMK.CLI");
--
--    will copy !USERS.JMK.CLI on M2 to !USERS.JMK.CLI on the
--    current machine.
--
--  Copy (Objects    => "!!M3!USERS.HJL.CLI.CMD",
--        Use_Prefix => "!USERS.JMK",
--        For_Prefix => "!USERS.HJL.CLI");
--
--    will copy the file !USERS.HJL.CLI.CMD on M3 to
--    !USERS.JMK.CMD on the current machine.
--    note when repositioning Objects it is necessary to give a
--    for_prefix which is a prefix of the Objects part of the
--    source parameter.
--
--  Copy (Objects    => "!!M1!USERS.HJL.ILFORD",
--        Use_Prefix => "!!M2!AGFA",
--        For_Prefix => "!USERS.HJL");
--
--    will copy !USERS.HJL.ILFORD from machine M1 to
--    machine M2 !AGFA!ILFORD
--
--  Copy (Objects    => "!USERS.HJL.CLI",
--        Use_Prefix => "!!M1",
--        Options    => "REPLACE AFTER=12/25/85");
--
--    will copy those files which have changed since 12/25/85 in
--    !USERS.HJL.CLI on the current machine to machine M1 !USERS.HJL.CLI
--    Any existing files with the same names will be overwritten.
```

```
    ------------------------------------------------------------------------

    procedure Server;

    -- start the archive server;

    procedure Status (For_Job : Machine.Job_Id);

    -- Prints information about the status of the Archive job specified.
    -- Can be the job number of an Archive Server or of a job running
    -- Archive.Copy, Archive.Restore, or Archive.Save.

    pragma Subsystem (Archive);
    pragma Module_Name (4, 3546);
end Archive;
```