

## ACTIVE-X DLL STRUCTURE

In the RSMS-4G architecture, the instrument, measurement, extraneous measurement, antenna pointing, and triggering routines will be implemented as ActiveX DLL components containing objects that may be instantiated at runtime. Each ActiveX DLL will contain three sections: 1) a public interface, 2) private data, methods, and objects, and 3) user interface forms.

**I. MEASUREMENT METHOD COMPONENT:** The measurement-method component will be implemented as shown in Figure 1 and sectioned into the three sections as follows:

**A. Private methods and data:**

1. **clsPreSelMgr:** each measurement public interface object can (but is not required to) instantiate their own copy of the class of type *clsPreSelMgr* and then call *Property Set SysConfigObj* to pass it a reference to an object of type *clsConfigPubInterface*. The object of type *clsPreSelMgr* is particularly useful for controlling a variety of preselectors without having to customize the code for each type of preselector. This also eliminates the need to recompile the measurement DLL code each time a new preselector is added to the repertoire of instruments.
2. **clsCallBack:** this class is contained in the *RSMS4Gtypelib.DLL*, the latter of which is referenced by the measurement DLL projects. A single object of this class will be instantiated and sent by reference to each instrument DLL in the signal path by sending it through the corresponding object of class *clsInstPubInterface* (using

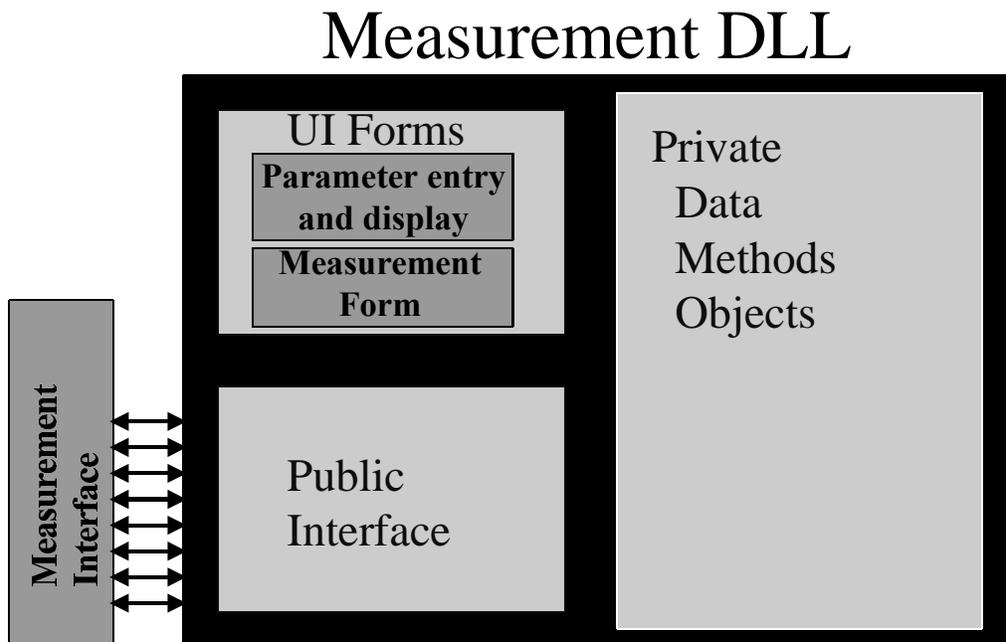


Figure 1

property *CallBackObj*). When it is deemed necessary to send an error message to the calling measurement DLL, the object of class *clsInstPubInterface* will call *Sub MeasError* of the Call-Back object. The object of class *clsCallBack* then raises an event and relays the error information, which is then trapped by the form *frmMeasForm* for further evaluation. A unique ID, which is the same as the *Access ID* described in Function *GetInstPubInterface* of class *clsConfigPubInterface*, is passed by argument to the object for which the *CallBack* object is passed. This is used by the instrument DLL to identify itself when calling certain subroutines within the object of type *clsCallBack*; this is so the instantiating object knows who made the call. It is therefore the responsibility of the measurement DLL to keep track of which ID is associated with which instrument.

A second object of type *clsCallBack* will be declared (but not instantiated) in the object of type *clsMeasPubInterface*. A reference to an object of type *clsCallBack* will be passed to the interface, from an executor, by calling Property *Set CallBackObj*. As discussed in the documentation for the *CallBack* class, this will be used to raise various event to the executor.

3. ***clsDocker***: this class is contained in the *RSMS4Gtypelib.DLL*, the latter of which is referenced by the measurement DLL projects; the class will be included (declared but not instantiated) in each of the measurement DLL projects. Its purpose is to provide instructions, when queried, as to where to dock the measurement form when placed in *Reduced Passive Display* mode. This object is vital to the operation of the measurementDLL and must be obtained by the measurement form prior to use. A single object of this class will be instantiated by the system configuration package and made available by reference to measurement DLLs by calling the Property *Get Docker()* in the object of type *clsConfigPubInterface*.
4. ***clsConfigPubInterface***: this class is contained in the *SysConfigLib.DLL*, the latter of which is referenced by the measurement DLL projects; the class will be included (declared but not instantiated) in each of the measurement DLL projects. Its purpose is to provide access to the different instruments within the signal path and to obtain information about those instruments.
5. ***clsCommon.cls***: this class is contained in the core program subdirectory and included in each of the measurement DLL projects.
6. ***Common.bas***: this Basic module is contained in the core program subdirectory and included in each of the measurement DLL projects.
7. ***Visa32.bas***: this Basic module is contained in the core program subdirectory and included in each of the measurement DLL projects.
8. ***clsMD5.cls***: provides an algorithm for creating an MD5 Hash Code from a byte array.

#### B. ***User Interface***:

1. **Parameter-Entry-and-Display Form**: (accessible only through public interface class *clsMeasPubInterface*, i.e., cannot be instantiated outside the DLL) provides a user interface for two usage modes:
  - a. ***Data-preview mode***: displays measurement parameter settings of measurement

data (of which the values cannot be changed).

- b. **Measurement-setup mode:** displays and allows editing of the current measurement parameter settings or entry of new parameters. Parameters are stored as local members and packaged as a byte array using a local parameter packaging/un-packaging method, the format of which is described in subroutine *MeasParamPackaging* within *clsMeasPubInterface*. In this mode, the user has the options of asserting the current settings, saving the settings to file, and recalling the settings from file. “Asserting the Current Settings” has different meaning depending upon the context upon which it is used; when setting up for a scheduled event, it means the parameters are packaged up and sent back to the schedule editor to be used later when executing an event; when setting up for an interactive-automated measurement, it means the parameters are sent back to the *Measurement Form* for implementation of the measurement procedure. **NEED TO MAKE A NOTE ABOUT TRAPPING THE EVENT SettingsChanged IN THE Callback CLASS.**

When this form is closed, it becomes hidden but remains instantiated. It will only be set to “Nothing” when the object of type *clsMeasPubInterface* is set to “Nothing”.

- 2. **Measurement Form:** (accessible only through public interface class *clsMeasPubInterface*, i.e., cannot be instantiated outside the DLL) contains methods for execution of the specified measurement. This form controls instruments via ActiveX-instrument components made known through a form of type *frmSystemConfig* (in the System Configuration Package). This form contains information about each discrete RF component and controllable instruments required to receive the signal, including information about proper bus connections. Access to individual instrument controls is accomplished by first calling *Property ComponentList* within the object of *clsConfigPubInterface* to obtain a list of components in the signal path. By using information in the component list, references to individual objects of *clsInstPubInterface* can be obtained, and in turn, references to the corresponding command/query objects is made available. References to the individual instrument public interfaces is obtained by calling Function *GetInstPubInterface* of class *clsConfigPubInterface*. **MAKE A NOTE ABOUT NOT USING vbModal AND ALLOWING CHANGE IN FOCUS FROM THE MEASUREMENT FORM.**

- a. **Display modes:** There are 3 different *display modes* for every measurement form.:
  - (1) *Full Interactive Display:* this has all of the features of the measurement form, including data display, control buttons, and indicators. This display can be minimized to the toolbar or put into “Reduced Passive Display” by right clicking on display
  - (2) *Reduced Passive Display:* this has the measurement data display but no control / query buttons or menus. It is reduced in size with the following dimensions:
    - (a) *Form:* Width: 2000, Height 1500

- (b) *Graph*: Width: 1800, Height: 1000

and docked at the edge of the main rms4g form. This is used to observe the instrument display without taking up as much screen real estate. This display can be minimized to the toolbar or can be returned to the default display mode by right clicking on display. There is minimal information displayed, but a title bar should be present to identify the measurement. Format for this form can be imported from a format file that is generated by the National Instruments software.

- (3) *Minimized*: The instrument form is minimized to the toolbar and will have an indicator of measurement progress (see below).
  - (a) When minimized, no commands sent to the virtual panel will be executed. All commands and queries must be executed by calling methods in the command/query object only.
  - (b) When minimized, the Virtual panel will also stop sending any commands to the instrument.

#### **DO I NEED TO HAVE FORM USAGE MODE SET THROUGH A SUBROUTINE.**

- b. *Usage modes*: The measurement form also provides a user interface for five usage modes:
  - (1) *Stealth Mode*: In this mode all of the classes and forms are instantiated but no forms are displayed. This is the default mode when the object of type *clsMeasPubInterface* is instantiated and is used primarily during the retrieval of data for the purpose of processing without displaying.
  - (2) *Data-preview Mode*: display previously acquired measurement data and information (during which the values cannot be changed and some buttons are disabled) Defaults to *Full Interactive Display* with specified controls, buttons, and menus disabled. Can be put in *Reduced Passive Display mode* or minimized.

At the onset of this mode, the object of type *clsMeasPubInterface* instantiates an object of type *clsCallBack* and passes a reference of the call-back object to each of the instruments in the signal path by calling the Property *Set CallBackObj* in the object of type *clsInstPubInterface*.
  - (3) *Interactive-Measurement Mode*: (Defaults to *Full Interactive Display mode* with the capability to put it into *Reduced Passive Display mode* or to minimize). displays the current settings, allows the user to enter new settings, and/or interactively control the execution of an automated/semi-automated measurement for the purpose of saving data. ***Upon completion of the automated measurement, all instrument access IDs are immediately relinquished, all references to objects of type clsInstPubInterface are set to "Nothing", and the data is saved to file at the users request but the form remains visible until closed by the***

~~*user. If the user wishes to perform a measurement again before closing the window, access to each instrument must once again be requested and granted prior to use of the instrument.*~~ In order to save the data, the form *frmIntAutExec* must be notified about measurement completion by calling subroutine *SaveMeasData()* in the object of *clsCallBack* (see UML sequence diagram - Figure 1.7). The interactive-automated executor then calls the Property *MeasData()* to obtain the measurement data. The call-back object is instantiated in the interactive-automated executor and passed by reference to the measurement DLL (not the same as the call-back object instantiated in the measurement DLL and passed by reference to the various instrument DLLs). Closure of the measurement is accomplished by asserting a “close” option (including “X” in the right upper corner of the window). In turn, the subroutine *CloseMeas()* of the object of type *clsCallBack* is called. The call-back object then raises the event *MeasShutDwn()* in the form *frmIntAutExec*, which in turn, deallocates the particular measurement public interface object (*clsMeasPubInterface*) by setting it to “Nothing”. Since only one measurement runs at a time, no ID is necessary.

If there is more than one instrument of the same type (e.g., two spectrum analyzers) in the signal path, a window is displayed so that the user can choose which instrument to use for the measurement.

At the onset of this mode, the object of type *clsMeasPubInterface* instantiates an object of type *clsCallBack* and passes a reference of the call-back object to each of the instruments in the signal path by calling the Property *Set CallBackObj* in the object of type *clsInstPubInterface*.

- (4) *Observation Mode*: allows the user to observe the data during event-scheduled measurements (during which some of the buttons are disabled, and parameters cannot be changed). Defaults to *Reduced Passive Display mode* with the capability to put into *Full Interactive Display mode* or to minimize.
- (5) *Event-Setup Mode*: allow the user to set up measurement parameters and then apply the settings when building an event. It can also be used when building an elaborated event to display the current parameter settings and to give the user the option of changing the current settings. In this mode, the measurement form is hidden and only the *Parameter-Entry-and-Display Form* is made visible. When the user is setting up an event in the event editor, each of the virtual panels in the signal path will be accessible and the user will set up each instrument and the measurement as desired. Then when the user saves the event to file, the editor queries the settings of each of the instrument DLLs, as well as, the measurement DLL. While placed in event-setup mode, any call to Property *Get*

*MeasData()* will bring up a message box saying “No data available”

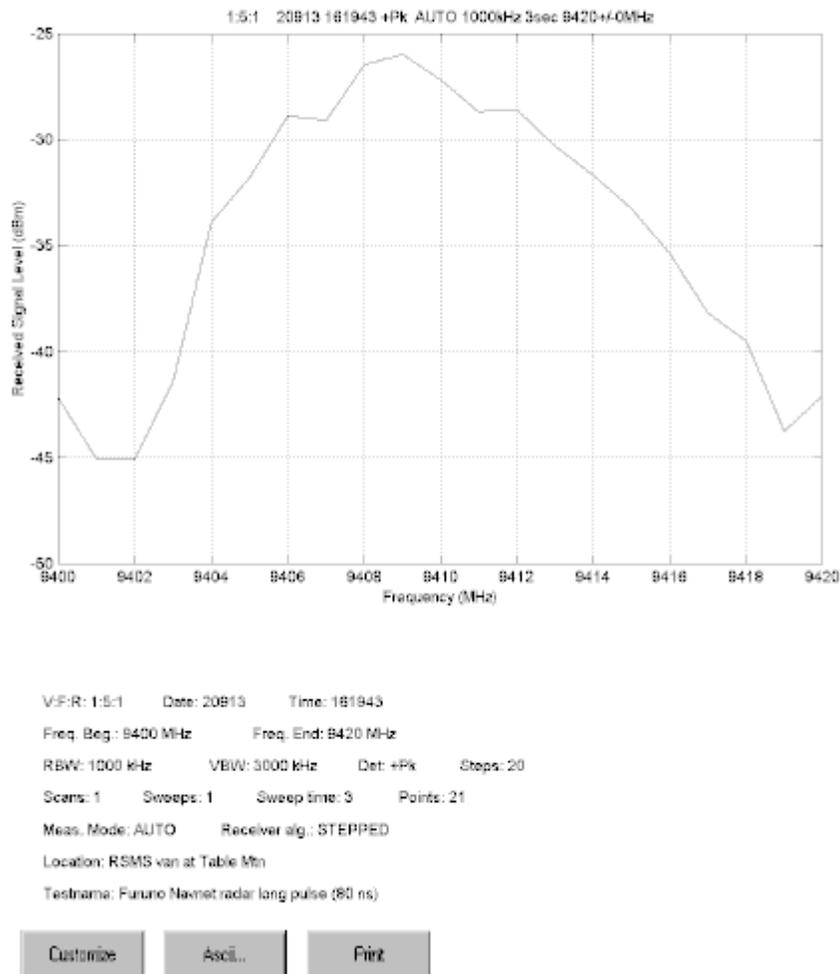
If there is more than one instrument of the same type (e.g., two spectrum analyzers) in the signal path, a window is displayed so that the user can choose which instrument to use for the measurement.

Prior to execution of a measurement, the measurement routine should query each instrument in the signal path as to whether it is in “dynamic” or “static mode”; this is accomplished by calling Property *Get StaticVsDyn()* in the object of *clsInstPubInterface*. A decision can then be made as to whether to grant the user (staff member) the right to have dynamic control of the instrument. If designated as “static”, the instrument, during automated measurements, stays in the setup state determined by the user, where the state is set either during schedule editing or prior to an interactive-automated measurement. If designated as “dynamic”, the measurement DLL is given the permission to alter the state of the instrument during execution of the measurement routine. There is no way to prevent the measurement routine from changing the state of the instrument even if it is designated as “static”, and therefore, this option can be overridden if necessary. Whether the instrument is to be designated as “static” or “dynamic” is determined at the time the user designates the signal path in the system configuration form. The user is automatically given this choice at the time the instrument is designated as being in the signal path.

- c. **Menu Options:** Standard top level menu items should be: “File”, “Edit”, “Setup”, “Show Settings”, “Tools” - in that order. The “File” menu will be organized as shown in the following diagram:



Besides the instrument control buttons and data-display, the *Measurement-Form* will have the following standard capabilities implemented as menu items (at the programmer discretion, buttons with the same functionality may also be included):



**Figure 3**

- (1) ***Print*** (contained in the *File* menu): Enabled only for the *Interactive-Measurement* mode, and the *Data-preview* mode, this option allows the user to print a report quality graph of the data as illustrated in the figure below. On the “File” menu (as illustrated above), there will be a “**Print**” caption. If the data is passed by the call to Property *Get MeasStream* in the object of type *clsMeasPubInterface*, this button will be disabled.
- (2) ***Save data record to ASCII file*** (contained in the *File* menu): writes the data to a ASCII formatted text file along with the appropriate labels (This option is only available in the *data-preview* mode). This is

accomplished by calling the Subroutine *Save2ASCII()* in the object of type *clsCallBack*. In turn, the call-back object raises the event *Dump2ASCII()*, which is trapped in either the form of type *frmSchedExecutor* or *frmNonSchedExecutor*. The executor then has each pertinent object write, to an ASCII file, the data for which it is responsible for packaging. On the “File” menu (as illustrated above), there will be an “**Export**” caption. If the data is passed by the call to Property *Get MeasStream* in the object of type *clsMeasPubInterface*, this button will be disabled.

- (3) **Measurement-parameters** (contained in the *Setup* menu): enabled only during the *data-preview* mode, the *interactive-measurement* mode, and the *observation* mode, this option is used to enter and/or examine measurement parameters using the *Parameter-Entry-and-Display Form*. On the “Setup” menu, there will be an “**Meas Parameters**” caption.
- (4) **Signal-path** (contained in the *Edit* menu): Enabled only in the *interactive-measurement mode* and *data-preview* mode, this feature is used to display and give focus to the form of type *frmSystemConfig* so that the user can examine the current system configuration and signal path; in *interactive-measurement* mode this request to see the signal path also allows the user to make changes if necessary and to designate individually whether each instrument (including preselector) is to be manually set (static mode) or fully automated (dynamic mode). When in *data-preview* mode, the System-Configuration-Form is also set in the same mode, and therefore, the system configuration and signal path cannot be changed. The request to see the signal path is accomplished by calling the subroutine *SetHrdwrCnfgAndPath()* in the object of type *clsConfigPubInterface* to show and set the focus of the form of type *frmSystemConfig*. If, while in the *interactive-measurement* mode, any change is made to the signal path (including change from *static* or *dynamic* mode), the object of *clsConfigPubInterface* raises the event *SigPathChanged()* which is then trapped by the form of type *frmNonSchedExecutor*. The interactive-automated executor then notifies, via subroutine *PathHasChngd()*, any active measurement DLL, extraneous-measurement DLL, and/or antenna-position-control DLL that the path has changed. Should this occur, each of the DLLs immediately relinquish all instrument Access IDs, each reference to an object of type *clsInstPubInterface* is set to “Nothing”, and then access rights are re-established for all necessary instruments for which control is required. On the “Edit” menu, there will be a “**Signal Path**” caption.

If more than one instrument of the same category is required for the measurement, then during measurement setup, the user will be queried

as to which instrument in the signal path is to be used for which operation.

- (5) **Re-measure** (contained in the *Tools* menu): Enabled only in the *data-display* mode, this option executes the measurement using the same parameters contained in the file being examined (provided the system hardware configuration is the same - i.e., the MD5 Hash Codes of the hardware configuration is the same as that stored in the signal-flow-path section of the data record). If the data was acquired through the scheduler, the measurement will be re-executed by calling up the scheduler using the same scheduler file and, highlighting the specific elaborated event and executing the specific event. This is made possible by the fact that the scheduler file and MD5 Hash Code of the elaborated event is stored in the data record. If the measurement was performed using the measurement form in the *interactive-measurement mode*, there is no scheduler file stored in the data file, and therefore, the measurement can be repeated using the measurement form in the same interactive mode. Re-measurement is accomplished by calling Subroutine *ReMeasure()* in the object of type *clsCallBack*. The call-back object then raises the event *DoReMeas()* which is trapped in either the form of type *frmSchedExecutor* or *frmNonSchedExecutor*. The executor then calls all the appropriate objects to re-measure with the same settings contained in the data record. On the “Tools“ menu, there will be a “**Remeasure**” caption.
- (6) **Start-, pause-, and abort-measurement buttons** (located as buttons on a toolbar as illustrated below): - Used to “start”, “pause”, “abort”, and/or



“abort and close” the measurement. (The “start” and “abort” options are only enabled in the interactive-measurement mode but the “pause”, and “abort and close” options are available in both the *interactive-measurement* modes and the *Observation Mode*). The “abort” command simply stops the measurement but allows the user to start anew in the same window; the “abort and close” command stops the measurement, deallocates the object of type *clsInstPubInterface*, and aborts (for that measurement) any intent to save data to file. In the “abort and close” case, “closing” is accomplished by calling Subroutine *CloseMeas()* in the object of *clsCallBack* (See UML sequence diagram -Figure 1.8). The call-back object raises the event *MeasShutDwn()* in either the form of type *frmSchedExecutor* or *frmNonSchedExecutor*, which in turn, deallocates the particular measurement public interface object

(*clsMeasPubInterface*) by setting it to “Nothing”. For scheduled measurement that has been aborted and closed, the scheduler then proceeds to the next measurement in the queue.

- (7) **Close (including “X” in right upper corner of the window):** this option will be disabled in all of the usage modes except *interactive-measurement mode* **and will only be enabled when the interactive-automated measurement is complete and the data has been written to file.** In the remaining cases, the forms will be closed as the object of *clsInstPubInterface* is set to “Nothing” by other objects or when the user asserts the “abort and close” option. The “close” option is implemented by a call to the subroutine *CloseMeas()* of the object of type *clsCallBack*. The call-back object then raises the event *MeasShutDown()* in the form *frmIntAutExec*, which in turn, deallocates the particular measurement public interface object (*clsMeasPubInterface*) by setting it to “Nothing”. The same procedure is implemented for the “abort and close” option, but in the case of the “close” option, the data is already saved to file.

### C. **Public Interface:**

1. **Public-Interface Class:** provides public access to forms, classes, and their associated methods and members. When instantiated, this object immediately instantiates the appropriate forms and automatically goes into *stealth mode*. Data can be sent to file and errors can be sent back to the forms of type *frmNonSchedExecutor* or *frmSchedExecutor* by calling the subroutine *MeasError()* in the object of type *clsCallBack*. This class will have a standard name called *clsMeasPubInterface* (instancing = “multiuse”). **MAKE A NOTE HERE ABOUT IMPLEMENTATION OF IGenMeasurment**

**Event Trapping:** Assuming the instantiation of an object “m\_CallBack” of type *clsCallBack* as follows:

**Private Withevents m\_CallBack As clsCallBack,**

the following subroutines are implemented to trap raised events:

- a. **Private Sub m\_CallBack\_SetSignalPath():** This is used to trap the event raised in the object of type *clsCallBack* requesting to see the signal path and system configuration. The object of type *clsMeasPubInterface* then calls *SetHrdwrCnfgAndPath* in the object of type *clsConfigPubInterface* to show and give focus to the form of type *frmSystemConfig*. (See UML sequence diagram - Figure 1.5)
- b. **Private Sub m\_CallBack SetInstError(ByRef ErrData As ErrUDT, ByVal vID As Integer):** This is used to trap the event relaying error information, raised in the object of type *clsCallBack*. The error information is passed via the user-defined type *ErrUDT* defined in object of type *clsCommon*. Once the event is trapped, it left to the object of type *clsMeasPubInterface* to determine

what type of action is to take place (i.e. disregard, resolve problem, try again, raise an event in the measurement executor, etc.) The argument "vID" is used to 'identify the particular instrument making the request and is the same ID received 'when passed the call-back object by calling Sub SetCallBackObj in the instrument 'interface object.

**Public Methods:** Class *clsMeasPubInterface* will have the following public methods:

- a. **Public Sub SetCallBackObj(ByRef vCallBackObj As clsCallBack, ByVal vID As Integer):** passes by reference an object of class *clsCallBack*. This object can be used to relay to the instantiating object such things as errors, and request to see the signal path. The argument *vID* is a unique ID given to the object for which the *CallBack* object is passed. This is used by the measurement DLL to identify itself when calling certain subroutines within the object of type *clsCallBack* so that the instantiating object knows who made the call. This object is vital to the operation of the instrument DLL and must be passed to it prior to use.
- b. **Public Property Set MsgHndlr(ByRef MsgHndlrObj As clsMessageHandler):** This property passes an object of type *clsMessageHandler* to the individual measurements so that messages can be displayed. Errors can be displayed but errors should also be passed through the object of type *clsCallBack* so that they can be handled properly.
- c. **Public Property Set SysConfigObj(ByRef vSysConfigObj As clsConfigPubInterface):** passes by reference an object of class *clsConfigPubInterface*. This object This object is vital to the operation of the instrument DLL and must be passed to it prior to use.
- d. **Public Function MeasName() As String:** returns a string that designates the name of the measurement (e.g., Stept, SweptM3, etc.)
- e. **Public Property Let AssocCals(ByRef CallList() As String):** This property sends an array of strings containing information about calibrations (or measurements) that are to be associated with the this measurement. If there are no calibrations (or measurements) to be associated with this measurement, the list is empty. Each string contains the user-defined calibration (or measurement) name and the DLL file name of the measurement or instrument (for data dumps) that was responsible for acquiring the associated data. The calibration (or measurement) name and the DLL name are separated by a semicolon and there should be no spaces after the semicolon. Example:

StepMeas 2.123-2.312 GHz;Stepped.dll

This subroutine is must be called by the measurement executor prior to requesting a measurement to be performed.

- f. **Public Subroutine PassCalCore (ByRef CoreData() As Byte, ByRef FrmtColmns as Integer, ByRef FrmtRows as Long, ByRef FrmtMode as Integer, ByRef FrmtVarType() as Integer, ByRef FrmtVarLabels() as VarLabels, ByRef FrmVarUnits As VarUnits) As Byte():** The subroutine passes, by argument, an array of bytes containing only the data - without a header - the purpose being to provide un-packaged data for use. The data will represent the actual value and will not require scaling and/or offset. The data will also be raw, meaning that it is not corrected by any calibration factor. All independent and dependent variables will have the same vector length, so that, for every value in the independent variable, there are corresponding values in each dependent variable. The array of bytes is formatted in a manner described by the variables in the argument list. These are as follows:
- (1) *FrmtColmns*: This is an integer returned by reference that designates the number of data variables (columns).
  - (2) *FrmtRows*: This is a long integer returned by reference that designates the number of data points for the variables (rows).
  - (3) *FrmtMode*: This is an integer returned by reference that designates the manner in which the variables are placed in the byte array. Mode = 0 designates a block mode in which all data of the first variable is put into the byte array, followed by all of the data of the second variable, etc. Mode = 1 designates interleaved mode in which the byte array is organized with the first data point of all of the variables, followed by the second data point of all of the variables, etc.
  - (4) *FrmtVarType*: This is an array of integer (array length equal to *FrmtColmns*) returned by reference that designates the variable type for the different columns of data. These variables are designated by the following:
    - (a) 1 = two byte integer
    - (b) 2 = four byte integer
    - (c) 3 = four byte IEEE floating point
    - (d) 4 = eight byte IEEE floating point
  - (5) *FrmtVarLabels*: This is an array of enumerated type *VarLabels* (defined in *clsCommon*) returned by reference that contains the column labels. Index "1" refers to column "1".
  - (6) *FrmtVarUnits*: This is an array of enumerated type *VarUnits* (defined in *clsCommon*) returned by reference that contains the units of the column. Index "1" refers to column "1". This function should only be called if the 10<sup>th</sup> item in the record header indicates that the record is "packaged", as apposed to "streamed".
- g. **Public Sub DoInteractiveMeas():** Setting to *interactive-measurement mode*,

this subroutine instantiates and shows the *Measurement Form* for interactive control of an automated measurement. Once the measurement is started by the user, it will run automatically until complete (or interrupted by the user). Once complete the routine will call subroutine *SaveMeasData()* in the object of type *clsCallBack*. (See UML sequence diagram - Figure 1.7) The call-back object will, in turn, raise an event to be trapped in form *frmIntAutExec*, The interactive-automated executor then calls Property *Get MeasData()* and Property *Get Params()* of the *clsMeasPubInterface* object for the purpose of obtaining the measurement data and parameters respectively. The form of type *frmMeasForm*, however, remains open until the user closes it. The “close” option is implemented by a call to the subroutine *CloseMeas()* of the object of type *clsCallBack*. The call-back object then raises the event *MeasShutDown()* in the form *frmIntAutExec*, which in turn, deallocates the particular measurement public interface object (*clsMeasPubInterface*) by setting it to “Nothing”. The same procedure is implemented for the “abort and close” option, but in the case of the “close” option, the data is saved to file prior to closing.

Property *Let AssocCals* in the object of type *clsMeasPubInterface* must be called prior to making a call to do a measurement.

Because the measurement may require loops that tie up the focus for long periods of time, the Visual Basic function *DoEvents* will be placed in these loops to allow branching to events as they occur.

- h. ***Public Sub DoSchdMeas(MeasParams() As Byte)***: Setting to *Observation mode*, this subroutine executes a scheduled measurement. The argument *MeasParams()*, which provides the parameter settings for the measurement, is passed as a byte array, originally packaged by the object of *clsMeasPubInterface* and obtained using the Property *Get Params()*. Contained within the header of *MeasParams* is a list of all instruments within the signal path which are to be controlled by this measurement. Any ambiguities with regard to multiple instruments of the same type will already have been resolved by the user when setting up the scheduled event. The serial number of the instrument is included in this information and can be used if there are two instrument of the same model type within the signal path.

Property *Let AssocCals* in the object of type *clsMeasPubInterface* must be called prior to making a call to do a measurement.

Once the measurement is complete, the subroutine will exit and return the focus to the calling routine. Thereafter, the measurement data can be obtained by calling Property *Get MeasData()* of the *clsMeasPubInterface* object.

- i. ***Public Sub EventSetup()***: Setting to *event-setup mode*, this subroutine opens the *Parameter-Entry-and-Display Form* so that the user can enter measurement parameters to be used for a scheduled event. When complete, the user asserts the apply-parameters option which simply closes the form but

keeps it instantiated. When the user asserts the “save event” option or the “save elaborated event” in the event editor or schedule editor respectively, a call to Property *Get Params()* of type *clsMeasPubInterface* is executed in order to get the parameters.

- j. *Public Sub DisplayData(ByRef AcqData() As Byte, ByRef ParameterSettings() As Byte)***: Setting to *data-preview mode*, this subroutine passes, to the interface object, the acquired data (*AcqData()*) and measurement parameters (*ParameterSettings()*) as arguments. The interface object then passes to the measurement form the data byte array, which is parsed into meaningful information, and displayed for examination. The *ParameterSettings* argument is passed to the *Parameter-Entry-and-Display Form*; this form is placed in *Data-preview mode* and then given the task of un-packaging the byte array to create meaningful information and to display the parameters to the user. For data gathered by calling Property *Get MeasStream*, the argument *AcqData* will have an array size of “1” containing a value of “0” and the Measurement form will display a message that says “Data streamed - not available for viewing.”
- k. *Public Sub ParseData(ByRef AcqData() As Byte, ByRef ParameterSettings() As Byte)***: Setting to *stealth mode*, this subroutine passes, to the interface object, the acquired data (*AcqData()*) and measurement parameters (*ParameterSettings()*) as arguments. The interface object then passes to the measurement form the data byte array, which is parsed into meaningful information. The *ParameterSettings* argument is passed to the *Parameter-Entry-and-Display Form* and then given the task of un-packaging the byte array to create meaningful information. The primary use for this subroutine is to interpret data into meaningful information that can be used for data processing by calling Function *GetCoreData* in the measurement public interface class. For data gathered by calling Property *Get MeasStream*, the argument *AcqData* will have an array size of “1” containing a value of “0” and the Measurement form will display a message that says “Data streamed - not available for viewing.”
- l. *Public Sub ParseParams(ByRef ParameterSettings() As Byte)***: Setting to *Event-Setup Mode*, this subroutine passes, to the interface object, the measurement parameters (*ParameterSettings()*) as an argument. In this *Event-Setup Mode*, the measurement form is hidden and only the *Parameter-Entry-and-Display Form* is made visible. The *ParameterSettings* argument (originally created when calling *Property Get Params*) is passed to the *Parameter-Entry-and-Display Form* and then given the task of un-packaging the byte array to create meaningful information. The primary use for this subroutine is to interpret parameter settings into meaningful information that can be used to set up measurement parameters and then apply the settings when building an event. It can also be used when building an elaborated event to display certain parameter settings from a previous measurement and to give

the user the option of changing the settings.

- m. *Public Property Get PkgType () As Integer*:** returns an integer which indicates the method by which data is to be passed. When *PkgType* = 0, the data is packaged as byte array and the executor should call *Property Get MeasData* in the object of type *clsMeasPubInterface* to retrieve the packaged data and, in turn, call Sub *WriteDataRecord* to pass the data on to the *File I/O Manager*. When *PkgType* = 1, the data, including header, is contained in a temporary holding file in which large quantities of data were streamed. In this latter case, the executor should retrieve the file name by calling *Property Get MeasStream* in the object of type *clsMeasPubInterface* and then pass it on to the *File I/O Manager* by calling Sub *WriteDataStream*.
- n. *Public Property Get MeasData() As Byte()*:** This property should be called by the executor only after calling *Property Get PkgType* in the object of type *clsMeasPubInterface* and receiving a return value of "0". The implementation of this property is mutually exclusive to *Property Get MeasStream*. In other words, only one of the two properties "*Get MeasStream*" and "*Get MeasData*" is implemented in agreement with *Property Get PkgType*. The other shows an error if called. This property returns, for either scheduled measurements or interactive-automated measurements, data packaged as a byte array with the following four required header entries located at the beginning of the array:

  - (1) A preamble containing the following characters: "RSMS4G\_DataPreamble". The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d) , and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the end.
  - (2) A version number to designate the version of the data packaging. There may be more than one way in which the data are packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
  - (3) The name of the Measurement ActiveX file responsible for packaging the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., Stepped.DLL )
  - (4) The number of bytes in the data package (including these three header entries). The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should

be preceded by zeros. (e.g., 0000532).

Following the header information are the data packaged according to the description provided by subroutine *DataPackaging* (documented in this section). At the end of the Data is a 32 bit CRC Code for the data only (to identify data corruption). Code for the CRC32 algorithm is located in Common.bas. The code takes an array of bytes as input and computes a 32-bit "checksum". To use the algorithm, call InitialiseCRC32tab and then call GetCRC32ForByteArray(Bytes).

- o. Public Property Get MeasStream() As String:*** This property should be called by the executor only after calling Property *Get PkgType* in the object of type *clsMeasPubInterface* and receiving a return value of "1". The implementation of this property is mutually exclusive to *Property Get MeasData*. In other words, only one of the two properties "*Get MeasStream*" and "*Get MeasData*" is implemented in agreement with Property *Get PkgType*. The other shows an error if called. This property returns, for either scheduled measurements or interactive-automated measurements, the full path and file name of a temporary file where the data has been streamed and which contains a header with the following four required entries located at the beginning of the file:

  - (1) A preamble containing the following characters:  
"RSMS4G\_DataPreamble". The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d) , and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the end.
  - (2) A version number to designate the version of the data packaging. There may be more than one way in which the data are packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
  - (3) The name of the Measurement ActiveX file responsible for packaging the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., Stepped.DLL )
  - (4) The number of bytes in the data package (including these three header entries). The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should

be preceded by zeros. (e.g., 0000532).

Following the header information are the data packaged according to the description provided by subroutine *DataPackaging* (documented in this section). At the end of the Data is a 32 bit CRC Code for the data only (to identify data corruption). Code for the CRC32 algorithm is located in Common.bas. The code takes an array of bytes as input and computes a 32-bit "checksum". To use the algorithm, call InitialiseCRC32tab and then call GetCRC32ForByteArray(Bytes).

- p. **Public Sub DataPackaging(ByRef PathAndFile As String):** Whether packaged as a byte array or streamed to a temporary holding file, this subroutine opens and append to an ASCII file designated by *PathAndFile* (full path and file name), the data packaging for each version of the particular measurement type. The format of the output will be as follows, where quotes designate a required title, strings inclosed in <> designate the value, and words in italics simply give an explanation and are not part of the output:

I) "MEASUREMENT-DATA PACKAGING" *this is a title for the section.*

A) " FOR:" *this is a subtitle - indented X 1.*

1) " Measurement Type: " <string> *where the sting represents a description of the measurement type - indented X 2.*

2) " Measurement-Data Packaging Version: " <value> *where "value" represents the version number of the measurement-data packaging - integers only - indented X 2*

3) " Date version originated: " <MM-DD-YY> *where MM= month, DD=day, YY=year, and must be represented by two digits for each value. (e.g., Date version originated 01-15-03) - indented X 2.*

B) "FORMAT:" *this is a subtitle - indented X 1*

1) "DATA HEADER:" *this is a subtitle - indented X 2*

" Header var #1 " <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

" Header var #2 " <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

" Header var #3 " <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

*etc., where type is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the header variable, description is a string describing what the variable*

represents, and **units** is a string describing the units.

2) "DATA:" *this is a subtitle - indented X 2*

a) "Number of Variables: " <value> where **value** represents the number of independent and dependent data variables - *indented X 3*.

b) *Description of the variables:*

" Variable #1 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

" Variable #2 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

" Variable #3 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

c) " Order " <sequential | alternated> where the order is either **sequential** (all of the values for variable #1 written first, then all the values for variable #2, etc ) or **alternated** (first value of all variables written first, followed by second value of all variables, etc.) - *indented X 3*

3) "DATA FOOTER:" *this is a subtitle - indented X 2*

" Footer var #1 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

" Footer var #2 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

" Footer var #3 " <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

---

### EXAMPLE:

MEASUREMENT-DATA PACKAGING

FOR:

Measurement Type: Swept M3

Measurement-Data Packaging Version: 1

Date version originated: 01-23-03

FORMAT:

DATA HEADER:

Header Var #1: Preamble; character bytes “RSMS4G\_DataPreamble”; bytes array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable;(none)

Header Var #2: version number; numerical characters terminated with a null character; variable;(none)

Header Var #3: Measurement ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #4: number of data bytes - including data header; data; and data footer; numerical characters terminated with a null character; variable;(none)

DATA:

Number of Variables: 3

Variable #1: Frequency; double; 8 bytes; (MHz)

Variable #2: Magnitude; double; 8 bytes; (dBm)

Variable #3: Phase; double; 8 bytes; (degrees)

Order: alternated

DATA FOOTER

Footer var #1: CRC of data; byte array; 4;(none)

MEASUREMENT-DATA PACKAGING

FOR:

Measurement Type: Swept M3

Measurement-Data PackagingVersion: 2

Date version originated: 06-10-03

FORMAT:

DATA HEADER:

Header Var #1: Preamble; character array “RSMS4G\_DataPreamble”; byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable;(none)

Header Var #2: version number; numerical characters terminated with a null

character; variable;(none)

Header Var #3: Measurement ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #4: number of data bytes - including data header; data; and data footer; numerical characters terminated with a null character; variable;(none)

#### DATA:

Number of Variables: 3

Variable #1: Frequency; double; 8 bytes; (MHz)

Variable #2: Real Component; double; 8 bytes; (dBm)

Variable #3: Imaginary Component; double; 8 bytes; (dBm)

Order: sequential

#### DATA FOOTER

Footer #1: CRC of data; byte array; 4;(none)

---

- q. *Public Sub MeasData2ASCII(ByRef DataVal() as Byte, ByRef FileAndPath As String)***: receives by reference, through argument *DataVal*, a byte array containing a measurement data (originally obtained through Property *Get MeasData()* from the object of type *clsMeasPubInterface*), un-packages the byte array to create meaningful information (based upon the version number), opens, for appending, the file designated by the argument *FileAndPath*, writes the measurement data along with appropriate labels to an ASCII file, and then closes the file. This subroutine will not be called if the original data was obtained by calling Property *Get MeasStream*.
- r. *Public Property Get Params() As Byte()***: returns the parameter settings for the measurement. The information is packaged as an array of bytes that can be written directly to an event file. At the beginning of the byte array, there are five required pieces of information contained in a header:
- (1) A preamble containing the following characters: “RSMS4G\_MeasParametersPreamble”. The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d) , and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the end.
  - (2) A version number to designate the version of parameter packaging. There may be more than one way in which the information is packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
  - (3) The name of the Measurement ActiveX file responsible for packaging

the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., Stepped.DLL )

- (4) An array of character bytes representing each of the instruments in the signal path **controlled** by this measurement (instruments for which access was granted by calling Function *GetInstPubInterface* in the object of type *clsConfigPubInterface*). Each instrument description is comprised of 4 fields separated only by a single comma. Each complete instrument description is separated by a semicolon and the last instrument is terminated with a null character. The 4 fields in each instrument description consists of the following (in this order):
  - (a) component model-name
  - (b) serial number
  - (c) category
  - (d) component ID.

For instance, the byte array may appear as follows:

HP8566,X23D14,Spec Analyzer,4;TEKTDS460,DS456X54,Oscilloscope,5 **i**

where **i** represents a null character.

Each of these 4 fields is obtained through the list of signal path instruments that is returned by calling Property *Get ComponentList* in the object of type *clsConfigPubInterface*.

If there are no instruments controlled by this measurements, the byte array consists of nothing more than a single null character.

- (5) The number of bytes in the byte array (including these five initial header entries). The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should be preceded by zeros. (e.g., 0000532).

This initial information is followed by the packaged parameter information formatted according to the description provided by the subroutine *MeasParamPackaging* (documented in this section).

- s. **Public Sub MeasParamPackaging(ByRef PathAndFile As String)**: opens and appends to an ASCII file designated by *PathAndFile* (full path and file name), the measurement-parameter packaging for the particular measurement type. The format of the output will be as follows, where quotes designate a required title, strings inclosed in <> designate the value, and words in italics simply give an explanation and are not part of the output:

I) “MEASUREMENT-PARAMETER PACKAGING” *this is a title for the section.*

A) “FOR:” *this is a subtitle - indented X 1*

1) “ Measurement Type: ” <string> *where the sting represents a description of the measurement type - indented X 2.*

2) “ Measurement-Parameters Packaging Version: ” <value> *where value represents the version number of the measurement-parameter packaging - integers only - indented X 2.*

3) “ Date version originated: ” <MM-DD-YY> *where MM= month, DD=day, YY=year, and must be represented by two digits for each value. (e.g., Date version originated 01-15-03) - indented X 2.*

B) “FORMAT:” *this is a subtitle - indented X 1.*

1) “MEASUREMENT-PARAMETER HEADER:” *this is a subtitle - indented X 2*

“ Header var #1 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

“ Header var #2 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

“ Header var #3 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

*etc., where type is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the header variable, description is a string describing what the variable represents, and units is a string describing the units.*

2) “MEASUREMENT PARAMETERS:” *this is a subtitle - indented X 2*

“Parameter #1: ” <description>; <type>; <# of bytes>;(<units>) - indented X 3

“Parameter #2: ” <description>; <type>; <# of bytes>;(<units>) - indented X 3

“Parameter #3: ” <description>; <type>; <# of bytes>;(<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

3) “MEASUREMENT PARAMETER FOOTER:” *this is a subtitle - indented X 2*

“ Footer var #1 ” <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

“ Footer var #2 ” <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

“ Footer var #3 ” <description>; <type>; <# of bytes>; (<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

---

### EXAMPLE:

#### MEASUREMENT-PARAMETER PACKAGING

FOR:

Measurement Type: Stepped

Measurement-Parameters-Packaging Version: 1

Date version originated: 04-15-03

FORMAT:

MEASUREMENT-PARAMETER HEADER:

Header Var #1: Preamble; character bytes “RSMS4G\_MeasParametersPreamble”; byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00) bytes; variable; (none)

Header Var #2: version number; numerical characters terminated with a null character; variable;(none)

Header Var #3: Measurement ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #4: An array of character bytes representing each of the instruments in

the signal path controlled by this measurement; Each instrument description is comprised of 4 fields (component model-name, serial number, category, and component ID) separated only by a single comma. Each complete instrument description is separated by a semicolon and the last instrument is terminated with a null character; variable;(none)

Header Var #5: number of data bytes - including header, parameters, and footer; Numerical characters terminated with a null character; variable;(none)

#### MEASUREMENT PARAMETERS:

Parameter #1: Spec. Analyzer Attenuation; integer; 2 bytes; (dB)

Parameter #2: Resolution BW; double; 8 bytes; (MHz)

Parameter #3: Video BW; double; 8 bytes; (MHz)

Parameter #4: Start Frequency; double; 8 bytes; (MHz)

Parameter #5: Stop Frequency; double; 8 bytes; (Hz)

#### MEASUREMENT-PARAMETER FOOTER:

None

#### MEASUREMENT-PARAMETER PACKAGING

##### FOR:

Measurement Type: Stepped

Measurement-Parameters Packaging Version: 2

Date version originated: 08-20-03

##### FORMAT:

#### MEASUREMENT-PARAMETER HEADER:

Header Var #1: Preamble; character array “RSMS4G\_MeasParametersPreamble”; byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable; (none)

Header Var #2: version number; numerical characters terminated with a null character; variable;(none)

Header Var #3: Measurement ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #4: An array of character bytes representing each of the instruments in the signal path controlled by this measurement; Each instrument description is comprised of 4 fields (component model-name, serial number, category, and component ID) separated only by a single comma. Each complete instrument description is separated by a semicolon and the last instrument is terminated with a null character; variable; (none)

Header Var #5: number of data bytes - including header, parameters, and footer; numerical characters terminated with a null character; variable; (none)

#### MEASUREMENT PARAMETERS:

Parameter #1: Analyzer Attenuation; integer; 2 bytes; (dB)

Parameter #2: Resolution BW; double; 8 bytes; (MHz)

Parameter #3: Video BW; double; 8 bytes; (MHz)

Parameter #4: Span; double; 8 bytes; (MHz)

Parameter #5: Center Frequency; double; 8 bytes; (Hz)

#### MEASUREMENT-PARAMETER FOOTER:

None

- 
- t. **Public Sub MeasParams2ASCII(ByRef ParamVal) as Byte,ByRef PathAndFile As String**: receives, through argument *ParamVal*, a byte array containing measurement parameter information (originally obtained through *Property Get Params* from the object of type *clsMeasPubInterface*), un-packages the array into meaningful information (based upon the version number), opens, for appending, the file designated by the argument *FileAndPath*, writes the measurement data along with appropriate labels to an ASCII file, and then closes the file.
  - u. **Public Property Get MeasParamMD5() As Byte()**: returns the 16 byte array containing the MD5 Hash Code for the measurement-parameter configuration - generated from the byte array obtained from *Property Get Params()* (excluding header information). The MD5 Hash code is obtained by calling the function *DigestByteArrayToHexStr* in the object of type *clsMD5*.
  - v. **Public Sub PathHasChngd()**: this subroutine informs the measurement DLL that the Signal Path and possibly the System Configuration has been changed. When this occurs the object of type *clsMeasPubInterface* relinquishes each of the instruments for which it has a valid Access ID, set to "Nothing" each reference to objects of type *clsInstPubInterface*, and then re-establish new Access IDs for each instrument for which control is required. Any time this subroutine is called, it also calls *Property Set SysConfigObj* in the object of type *clsPreSelMgr* which then automatically re-establishes the preselector configuration.
  - w. **Public Sub KillMeas()**: aborts the measurement. This is accomplished by setting a variable in the measurement form which designates that any loop should be exited and the measurement shut down gracefully. (See UML sequence diagram - Figure 2.2 and Project "Kill Program", the latter located in subdirectory: *c:\rsms4g\UML\TestCode\ToTestKillMeas.*)
  - x. **Public Property Let MeasFormVisibility(ByVal VisVal as Integer)**: sets the visibility of the form of type *frmMeasForm*, where 0 = hide (remains

instantiated), 1 = visible.

- y. **Public Property Get MeasFormVisibility() as Integer**: returns an integer designating whether the form of type *frmMeasForm* is currently visible or not. 0 = not visible, 1 = visible.

#### WE SHOULD ADD SOMETHING LIKE THE FOLLOWING FOR GETTING THE MEASUREMENT SETUP PARAMETERS

- z. **Public Sub PassCoreData(ByRef MeasObj as Object)** pass the particular calibration (or other data) to the measurement object. The argument *MeasObj* is an object of type *IGenMeasurement*.

- aa. **Public Function GetCoreData(ByRef FrmtColmns as Integer, ByRef FrmtRows as Long, ByRef FrmtMode as Integer, ByRef FrmtVarType() as Integer, ByRef FrmtVarLabels() as VarLabels, ByRef FrmVarUnits() As VarUnits) As Byte()**: This function returns an array of bytes containing only the data - without a header - the purpose being to provide un-packaged data for data processing routines. The data will represent the actual value and will not require scaling and/or offset. The data will also be raw, meaning that it is not corrected by any calibration factor. All independent and dependent variables will have the same vector length, so that, for every value in the independent variable, there are corresponding values in each dependent variable. The array of bytes is formatted in a manner described by the variables in the argument list. These are as follows:

- (1) *FrmtColmns*: This is an integer returned by reference that designates the number of data variables (columns).
- (2) *FrmtRows*: This is a long integer returned by reference that designates the number of data points for the variables (rows).
- (3) *FrmtMode*: This is an integer returned by reference that designates the manner in which the variables are placed in the byte array. Mode = 0 designates a block mode in which all data of the first variable is put into the byte array, followed by all of the data of the second variable, etc. Mode = 1 designates interleaved mode in which the byte array is organized with the first data point of all of the variables, followed by the second data point of all of the variables, etc.
- (4) *FrmtVarType*: This is an array of integer (array length equal to *FrmtColmns*) returned by reference that designates the variable type for the different columns of data. These variables are designated by the following:
  - (a) 1 = two byte integer
  - (b) 2 = four byte integer
  - (c) 3 = four byte IEEE floating point
  - (d) 4 = eight byte IEEE floating point

- (5) *FrmtVarLabels*: This is an array of enumerated type *VarLabels* (defined in *clsCommon*) returned by reference that contains the column labels. Index "1" refers to column "1".
- (6) *FrmtVarUnits*: This is an array of enumerated type *VarUnits* (defined in *clsCommon*) returned by reference that contains the units of the column. Index "1" refers to column "1".

II. **INSTRUMENT COMPONENT:** The instrument component will consist of 3 basic unit and implemented as shown in Figure 2. The three units described as follows:

A. **Private methods and data:**

1. **Command/query class:** (instancng = “PublicNotCreatable”) This class will be automatically instantiated upon instantiation of the public interface class *clsInstPubInterface*. It contains specific commands and query methods for the purpose of controlling an instrument and/or obtaining information or data; these methods are private and represent the implementation of one or more abstract command/query interface classes. Access to these private methods is made available by declaring the appropriate interface class and assigning, by reference, the command/query object to the interface object. Access to the command/query object is made available through the public method *GetCommQueryObj* contained within the public interface class (*clsInstPubInterface*). The command/query object should be passed a copy of the object of type *clsCallBack*; this object can be used to pass back errors to the client (a measurement executor) and to indicate changes in an instrument setting. For some instrument settings, it is best not to notify the callback object every time a change is made unless indicated by the client that doing so is

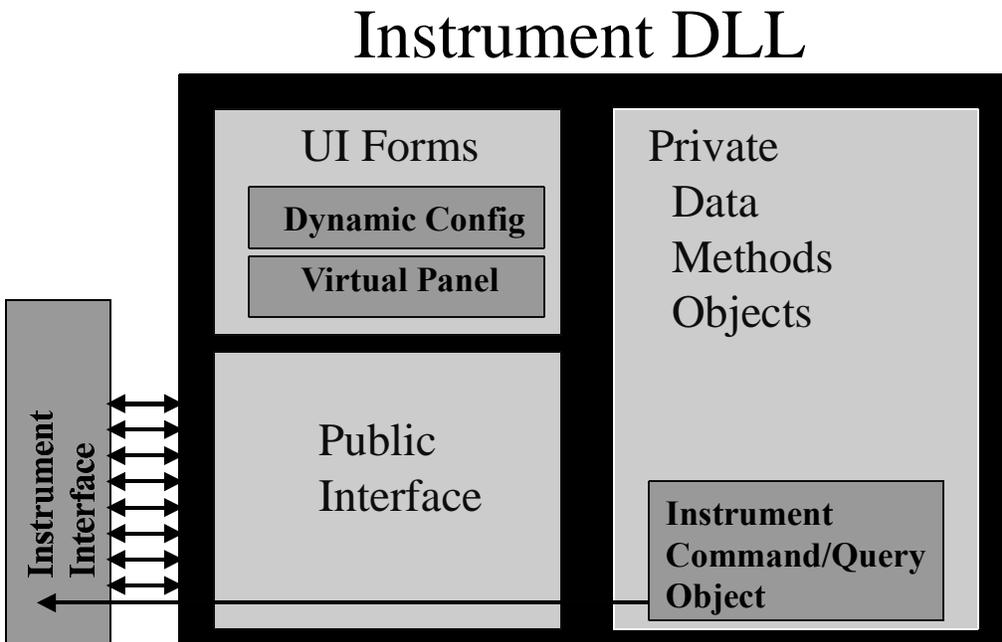


Figure 5

desirable. To do so may significantly slow down the measurement process. Instead, it may be better to have a subroutine call or an argument in the parameter list that can turn on or off the “settings-changed” notification process for those settings which may be called frequently during a measurement routine. For those settings that are not likely to be called frequently and repetitively, it is desirable to notify the callback object any time the setting is changed. Any time the setting is changed through the virtual panel, the “settings-changed” notification should (must) be sent through the callback object; this allows the measurement DLL to know of any user initiated change.

The following list show, for different instrument type, the interface class that will be included in the project and put in an “Implements” statement at the top of the command/query class.

- a. Spectrum analyzers: IGenSpeAnlZr.cls
  - b. Oscilloscopes IGenOscope.cls
  - c. Preselectors IGenPreSel.cls
2. **Visa32.bas**: this Basic module is contained in the core program subdirectory and included in each of the instrument DLL projects.
  3. **VISA.bas**: this Basic module is contained in the core program subdirectory and included in each of the instrument DLL projects.
  4. **clsCommon.cls**: this class is contained in the core program subdirectory and included in each of the instrument DLL projects.
  5. **Common.bas**: this Basic module is contained in the core program subdirectory and included in each of the instrument DLL projects.
  6. **clsDocker**: this class is contained in the *RSMS4Gtypelib.DLL*, the latter of which is referenced by the measurement DLL projects; the class will be included (declared but not instantiated) in each of the instrument DLL projects. Its purpose is to provide instructions, when queried, as to where to dock the virtual panel form when placed in *Reduced Passive Display* mode. This object is vital to the operation of the instrument DLL and must be provided to the instrument DLL by passing a reference to the Docker object prior to use of the instrument virtual panel. A single object of this class will be instantiated by the system configuration package and passed by reference to instrument DLLs by calling the *Property Set Docker()* in the object of type *clsInstPubInterface*; this will occur immediately after instantiating the object of type *clsInstPubInterface*.
  7. **clsCallBack**: this class is contained in the *RSMS4Gtypelib.DLL*, the latter of which is referenced by the measurement DLL projects; the class will be included (declared but not instantiated) in each of the instrument DLL projects. A single object of this class will be instantiated and sent by reference to each instrument DLL in the signal path by sending it through the corresponding object of class *clsInstPubInterface* (using property *CallBackObj*). When it is deemed necessary to send an error message to the calling measurement DLL, the object of class *clsInstPubInterface* will call *Sub MeasError* of the Call-Back object. The object of

class *clsCallBack* then raises an event and relays the error information, which is then trapped by the form *frmMeasForm* for further evaluation. A unique ID, which is the same as the *Access ID* described in Function *GetInstPubInterface* of class *clsConfigPubInterface*, is passed by argument to the object for which the *CallBack* object is passed. This is used by the instrument DLL to identify itself when calling certain subroutines within the object of type *clsCallBack*; this is so the instantiating object knows who made the call. It is therefore the responsibility of the measurement DLL to keep track of which ID is associated with which instrument.

The callback object can also be used to indicate changes in an instrument setting by calling the *SettingsChanged()* subroutine. For some instrument settings, it is best not to notify the callback object every time a change is made unless indicated by the client that doing so is desirable. To notify callback every time a setting is changed may significantly slow down the measurement process. Instead, it may be better to have a subroutine call or an argument in the parameter list that can turn on or off the “settings-changed” notification process for those settings which may be called frequently during a measurement routine. For those settings that are not likely to be called frequently and repetitively, it is desirable to notify the callback object any time the setting is changed. Any time the setting is changed through the virtual panel, the “settings-changed” notification should (must) be sent through the callback object; this allows the measurement DLL to know of any user initiated change.

8. ***ClsMD5.cls***: provides an algorithm for creating an MD5 Hash Code from a byte array.
- B. ***User Interface***: provides the user interface for displaying the instrument front panel and to examine instrument dynamic configuration.
1. ***Virtual-panel form***: (accessible only through public interface class *clsInstPubInterface*, i.e., cannot be instantiated or passed by reference outside the DLL) used to display controls, collect traces, and display the data, giving the appearance of the instrument front panel. **Any communication with the physical instrument will not be done until the usage is set through Property *Let VPMode* in the object of type *clsInstPubInterface*.** The virtual panel should never be placed in *vbModal* state so that users can change focus from virtual panel form to other forms.
    - a. ***Display Modes***: There should be 4 different ***display modes*** for every virtual panel. Once an instrument is assigned to the signal path (in the system configuration form) the virtual panel is instantiated and shown and cannot be hidden or deallocated until it is removed from the signal path (this means that the “X” in the right upper corner must be removed, as well as, any “close” buttons):
      - (1) ***Full Interactive Display***: this has all of the features of the virtual panel,

including data display, control buttons, indicators, and real-time control of the instrument. This display can be minimized to the toolbar or put into “Reduced Passive Display” by right clicking on display

- (2) *Reduced Passive Display*: this has the data display but no control / query buttons or menus. It is reduced in size with the following dimensions:
  - (a) *Form*: Width: 2000, Height 1500 (twips)
  - (b) *Graph*: Width: 1800, Height: 1000 (twips)

and docked at the edge of the main *rsms4g* form. This is used to observe the instrument display without taking up as much screen real estate. This display can be minimized to the toolbar or can be returned to the default display mode by right clicking on display. There is minimal information displayed, but a title bar should be present to identify the device during *Instrument-observation mode*. Format for this form can be imported from a format file that is generated by the National Instruments software.

- (3) *Passive Display*: this has all of the control buttons but no data display. When controls are asserted, none of the commands or queries are sent to instruments but the instrument virtual state is maintained. (“Virtual state” is the state of the instrument had the commands actually been sent.). This display is used for designating the setup when editing scheduled events. It also can be minimized.
- (4) *Minimized*: The virtual panel is minimized to the toolbar and will have an indicator of measurement progress during *Instrument-observation mode* (see below).
  - (a) When minimized, the call to *clsInstPubInterface* subroutines *UpdateVPTrace()* and *UpdateVPParam()* are not executed. All commands and queries must be executed by calling methods in the command/query object only.
  - (b) When minimized, the Virtual panel will also stop sending any commands to the instrument.

**b. Usage Modes:** The virtual panel will have six different *use modes* (all of which may have multiple *display modes*) as follows:

- (1) *Stealth mode*: In this mode all of the classes and forms are instantiated but no forms are displayed. This is the default mode when the object of type *clsInstPubInterface* is instantiated and is used primarily during the retrieval of data for the purpose of processing without displaying.
- (2) *Fully-manual mode*: allows the user to set up the instrument using remote control and then acquire and save data to a data file. Defaults to *Reduced Passive Display mode* with the capability to put in *Full Interactive Display mode* or to minimize.

When the virtual panel is instantiated and placed in this mode (but only after being placed in this mode), it will take on the settings of the device without changing any of the parameters (except possibly when it is necessary to emulate a function - things like persistence, auto-trigger, normal-trigger, averaging, etc.) but no data will be displayed on a trace until the user designates to do so.

- (3) *Immediate-instrument-setup mode*: allows the user to manually configure an instrument, in real time; this occurs when the specified instrument has been designated for manual control during a interactive-automated measurement using the *Measurement form (Measurement-Methods DLL)* or when designating the signal path. For example, the user may want to manually set a spectrum analyzer used as a down-converter and then automate the acquisition of data from a digital oscilloscope used to digitize the 2<sup>nd</sup> IF of the spectrum analyzer; in this case, the semi-automated measurement controls the digital oscilloscope using the interactive measurement form but leaves static setup of the spectrum analyzer to the user. Defaults to *Reduced Passive Display mode* with the capability to put in *Full Interactive Display mode* or to minimize.

When the virtual panel is instantiated and placed in this mode (but only after being placed in this mode), it will take on the settings of the device without changing any of the parameters (except possibly when it is necessary to emulate a function - things like persistence, auto-trigger, normal-trigger, averaging, etc.) but no data will be displayed on a trace until the user designates to do so.

- (4) *Recorded-instrument-settings mode*: allows the user to manually configure an instrument and save the instrument state (or virtual state) be used later for semi-automated event-scheduled measurements; this is used when the specified instrument has been designated for static control during a scheduled measurement. For example, when setting up an elaborated event using the schedule editor, the user may want to manually set a spectrum analyzer used as a down-converter, whereby the 2<sup>nd</sup> IF is fed to a digital oscilloscope for digitization. During the editing of the schedule event, the user goes to the spectrum-analyzer virtual-panel, sets up the device as desired, and places the variables describing the instrument state in an elaborated event to be used during scheduled execution. During execution, the variables describing the instrument state are sent to the spectrum analyzer for a static setup and then the measurement module automates the acquisition of data from a digital oscilloscope.

When the user is setting up an event in the event editor, each of the virtual panels in the signal path will be accessible and the user will set up each instrument and the measurement as desired. Then when the user

saves the event to file, the editor queries the settings of each of the instrument DLLs, as well as, the measurement DLL.

The recorded-instrument-setup mode can further be divided into two sub-modes as follows:

- (a) *Active*: in this sub-mode, the manual commands are physically sent to the instrument to make hardware changes as the user changes the various parameters on the virtual panel; therefore a physical connection via the control bus is required. Defaults to *Reduced Passive Display mode* with the capability to put in *Full Interactive Display mode* or to minimize.

When the virtual panel is instantiated and placed in this mode (but only after being placed in this mode), it will take on the settings of the device without changing any of the parameters (except possibly when it is necessary to emulate a function - things like persistence, auto-trigger, normal-trigger, averaging, etc.) but no data will be displayed on a trace until the user designates to do so.

- (b) *Passive*: in this sub-mode, the manual variables describing the instrument virtual state are simply recorded without sending the commands to the instrument; the control bus, therefore, does not have to be physically connected. Defaults to *Passive Display* with the capability to minimize.
- (5) *Instrument-observation mode*: allows the user to observe the display data trace during automated or semi-automated measurements. In this case, the control commands are sent to the command/query object, and “Update Data Trace” sent to the virtual panel. Defaults to *Reduced Passive Display* with the option to minimize or place in *Full Interactive Display* with controls, buttons, and menus disabled. When minimized the virtual panel no longer sends any command to the command/query object (i.e. ignores any calls to `clsInstPubInterface` subroutines `UpdateVPTrace()` and `UpdateVPPParam()`).

When the virtual panel is instantiated and placed in this mode (but only after being placed in this mode), it will take on the settings of the device without changing any of the parameters (except possibly when it is necessary to emulate a function - things like persistence, auto-trigger, normal-trigger, averaging, etc.) but no data will be displayed on a trace until the measurement calls `Sub UpdateVPTrace()` in the object of type `clsInstPubInterface`.

- (6) *Data-preview mode*: allows the user to examine previously acquired data from a fully-manual measurement. Defaults to *Full Interactive Display* with specified controls, buttons, and menus disabled. Can be put in *Reduced Passive Display mode* or minimized.

(7) *Setup-preview mode*: allows the user to examine the instrument setup from data previously acquired using either a *scheduled* or *interactive-automated* measurement. In this mode, the virtual panel is hidden and only the form of type *frmDynamicConfiguration* (minimized to the toolbar) is available to the user.

c. **Menu Options**: Standard top level menu items should be: “File”, “Edit”, “Setup”, “Show Settings”, “Tools” - in that order. The “File” menu will be organized as shown in the following diagram:



Besides the instrument control buttons and data-display, the virtual panel will have the following standard capabilities implemented as menu items (at the programmer discretion, buttons with the same functionality may also be included):

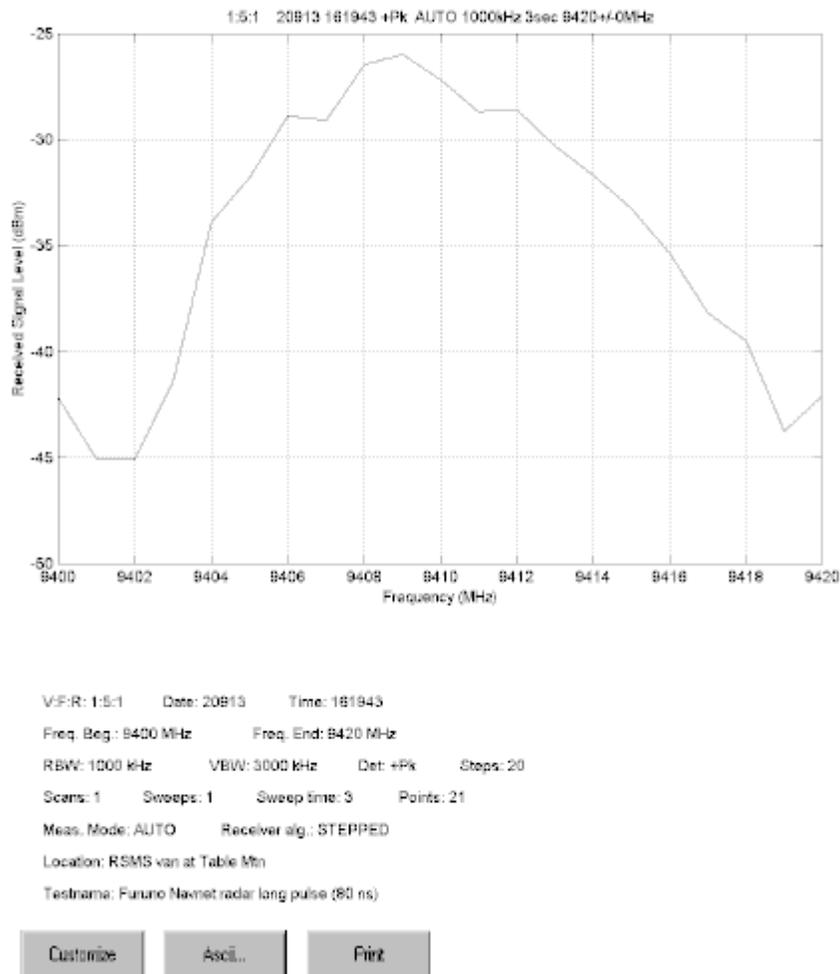


Figure 7

- (1) **Print** (contained in the *File* menu): Enabled only for the *fully-manual* mode, and the *Data-preview* mode, this option allows the user to print a report quality graph of the data as illustrated in the figure below. On the “File” menu (as illustrated above), there will be a “**Print**” caption. If the data is passed by the call to Property *Get DataStream* in the object of type *clsInstPubInterface*, this button will be disabled.
- (2) **Save-data** (contained in the *File* menu): Enabled only for *fully-manual* mode, this option is used to save the data, as shown on the display, to the current binary data file. This option is implemented by calling the Subroutine *SaveTrace* in the object of type *clsCallBack* (see UML sequence diagram - Figure 2.1). The call-back object then raises the

event *GetDataDump* which is trapped in form *frmFullManualExec*. This, in turn, results in a call to Property *Get DataDump()* of the *clsInstPubInterface* object for the purpose of obtaining the trace data from an instrument DLL. An ID argument passed in subroutine *Property Set CallbackObj*, which is the same as the *Access ID* described in Function *GetInstPubInterface* of class *clsConfigPubInterface*, is passed back as an argument in subroutine *SaveTrace* and is used to identify the particular instrument making the request. On the “File” menu (as illustrated above), there will be a “**Save**” caption.

- (3) **Save data record to ASCII file** (located in the file menu): Enabled only in the *data-preview* mode, this option writes the data to a ASCII formatted text file along with the appropriate labels. This is accomplished by calling the Subroutine *Save2ASCII()* in the object of type *clsCallBack* (see UML sequence diagram - Figure 2.0). In turn, the call-back object raises the event *Dump2ASCII()*, which is trapped in either the form of type *frmIntAutExec*, or *frmFullManualExec*. In turn, the executor has each pertinent object write, to an ASCII file, the data for which it is responsible for packaging. On the “File” menu (as illustrated above), there will be an “**Export**” caption. If the data is passed by the call to Property *Get DataStream* in the object of type *clsInstPubInterface*, this button will be disabled.
- (4) **Save/Recall Configuration** (located in the file menu): these options provides the user the option of saving the current instrument state to a file or to recall the state from file to configure the instrument. These option will be enabled only for the *Fully-manual mode*, the *Immediate-instrument-setup mode*, and the *Recorded-instrument-settings mode*. On the “File” menu (as illustrated above), there will be an “**Open Config**” caption and a “**Save Config**” caption. The “**Open Config**” should be disabled for the Passive *Recorded-instrument-settings mode*.
- (1) **Local/Remote** (contained in the *Setup* menu): Enabled only for *fully-manual* mode and *immediate-instrument-setup mode*, this option allows the user to put the instrument in local mode so that settings can be changed on the actual instrument panel.
  - (a) When possible, the instrument should be put the instrument in “lockout” at all times except when placed in local mode.
  - (b) There should be some indicator on the Virtual Panel to show that the device is in “local.”
  - (c) When the user puts the device back into remote mode (by either asserting the local/remote option in the menu or by trying to change one of the instrument settings from the virtual panel), the virtual panel does an automatic update of the instrument parameters.

- (d) On the “Setup” menu, there will be (with a check **U**capability) a “**Local Lockout**” caption on the pull down menu (i.e. when checked, it is in local lockout mode- the default state; when not checked, it is in local)
- (2) **Preset** (contained in the *Setup* menu): Enabled only for *fully-manual* mode, *immediate-instrument-setup* mode, and *recorded-instrument-setup* mode, this option allows the user to return the instrument to a preset state. On the “Setup” menu, there will be a “**Preset**” caption.
- (3) **Signal-path:** (contained in the *Edit* menu) Enabled only in the *fully-manual mode*, the *immediate-instrument-setup mode* or the *data-preview mode*, this feature is used to display the form of type *frmSystemConfig* so that the user can examine the current system configuration and signal path. In the in the *fully-manual mode* and *immediate-instrument-setup* modes this request to see the signal path also allows the user to make changes if necessary and to designate individually whether each instrument (including preselector) is to be manually set (static mode) or fully automated (dynamic mode). (See UML sequence diagram - Figure 1.6) When in *data-preview* mode, the System-Configuration-Form is also set in the same mode, and therefore, the system configuration and signal path cannot be changed. The request to see the signal path is accomplished as described in Property *Let VPMODE* of *clsInstPubInterface*. On the “Edit” menu, there will be a “**Signal Path**” caption.
- (4) **Re-measure** (contained in the *Tools* menu): Enabled only in the *data-preview* mode, this option informs the form of *frmNonSchedExecutor* which then sets up the measurement system to do measurement using the same parameters contained in the file being examined (provided the system hardware configuration is the same - i.e., the MD5 Hash Codes of the hardware configuration is the same as that stored in the signal-flow-path section of the data record). This is accomplished by calling Subroutine *ReMeasure()* in the object of type *clsCallBack*. The call-back object then raises the event *DoReMeas()* which is trapped in the form of type *frmNonSchedExecutor*. The executor then calls all the appropriate objects to re-measure with the same settings contained in the data record. On the “Tools“ menu, there will be a “**Remeasure**” caption.
- (5) **Close (including “X” button in right upper corner of the window):** this option will be disabled in all of the usage modes. The forms will be closed only as the object of *clsInstPubInterface* is set to “Nothing” by other objects

Because there may be loops that tie up the focus for long periods of time, the Visual Basic function *DoEvents* will be placed in these loops to allow branching to events as they occur.

2. **Dynamic Configuration form:** (accessible only through public interface class *clsInstPubInterface*, i.e., cannot be instantiated outside the DLL) provides a display of dynamic parameter settings. This will be used to display the current settings and to display instrument settings when examining data from a data file . The user is not allowed to make any changes from this form.

**B. Public Interface:**

1. **Public Interface Class:** provides public access to forms, classes, and their associated methods and members. Upon instantiation and when the mode has been set by calling the Property *Let VPMODE*, the virtual panel will be instantiated and shown, except for *Stealth mode* and *Setup-preview mode*, in which case, it is hidden; the public interface object will also immediately instantiate the *Dynamic Configuration* form (for all usage modes), and the *Command/Query* object (only for *Fully-manual mode*, *Immediate-instrument-setup mode*, *Active recorded-instrument-settings mode*, and *Instrument-observation mode*); upon termination, the *Public Interface* object also sets to “Nothing” any instantiated *Virtual Panel* form, *Dynamic Configuration* form, and/or *Command/Query* object. Data can be sent to file and errors can be sent back to the forms of *frmNonSchedExecutor* by calling the appropriate subroutine in the object of class *clsCallBack*. When a request for data-save is made via the object of *clsCallBack*, the Property *Get DataDump()* is, in turn, called to obtain the data.

The public interface class should implement all of the subroutines, functions, and properties contained in the class of *IgenInstrument*. This is enforced by placing the statement “*Implements IgenInstrument*” at the top of the code contained within the Public Interface class. The following is an example of a subroutine located in the class of *IgenInstrument* that must be implemented in the Public Interface class:

**Public Sub SetCallBackObj(ByRef vCallBackObj As clsCallBack, ByVal vID As Integer)**

‘There is no code contained within this subroutine

**End Sub**

The subroutine is implemented in the Public Interface class through the following code:

**Private Sub IGenInstrument\_SetCallBackObj(vCallBackObj As RSMS4Gtypelib.clsCallBack, ByVal vID As Integer)**

‘Code contained within this subroutine implements the subroutine

**End Sub**

Note that, when implemented, the subroutine is declared Private. (This is very

important.) The subroutine name is also preceded by “IGenInstrument\_” which designates the interface class that is being implemented.

The instrument public interface class will have a standard name called *clsInstPubInterface* (instanting = “multiuse”) and will have the following public methods:

- a. ***Public Sub SetCallbackObj(ByRef vCallbackObj As clsCallback, ByVal vID As Integer)***: passes by reference an object of class *clsCallback*. This object can be used to relay to the instantiating object such things as errors, trace save, request to see the signal path, and command-set save. The argument *vID*, which is the same as the *Access ID* described in Function *GetInstPubInterface* of class *clsConfigPubInterface*, is a unique ID given to the object for which the *Callback* object is passed. This is used by the instrument DLL to identify itself when calling certain subroutines within the object of type *clsCallback* so that the instantiating object knows who made the call. This object is vital to the operation of the instrument DLL and must be passed to it prior to use.
- b. ***Public Property Set Docker(ByRef DockerObj As clsDocker)***: passes by reference an object of class *clsDocker*, which is used to give instructions, when queried, as to where to dock the virtual panel when placed in *Reduced Passive Display* mode. This object is vital to the operation of the instrument DLL and must be passed to it by the system configuration package prior to use.
- c. ***Public Sub ShowVP()***: This subroutine shows the virtual panel when called but is ignored in the “*stealth*” usage mode.
- d. ***Public Property Let InstSessionID(ByVal IDVal as Long)***: passes the VISA instrument session ID. This must be done prior to using any virtual panel or sending messages to the command/query object and is accomplished by the System Configuration package.
- e. ***Public Sub ShutDown()***: This notifies the instrument DLL that the instrument interface is about to be deallocated. This gives the instrument a chance to do any cleanup work before deallocation. For example, an instrument that is in “local lockout” may need to be placed in “local” mode so that the user does not have to re-cycle the power on the device once software control is discontinued.
- f. ***Public Property Let StaticVSDyn(ByVal StateArg as Integer)***: designates whether the instrument is to be dynamic or static during an automated measurement - designated by the argument *StateArg*, where 0 = dynamic and 1 = static.. If designated as “static”, the instrument stays in the setup state determined by the user, where the setup state is set either during schedule editing or prior to an interactive-automated measurement. If designated as

“dynamic”, a measurement DLL is given the permission to alter the setup state of the instrument during execution of the measurement routine. There is no way to prevent the measurement routine from changing the setup state of the instrument even if it is designated as “static”, and therefore, this option can be overridden if necessary. However, the measurement routine should query for this setting using the Property *Get StaticVsDyn()*, so that a decision can be made as to whether to grant the user the right to have static control of the instrument. Whether the instrument is to be designated as “static” or “dynamic” is determined at the time the user designates the signal path in the system configuration form. The user is automatically given this choice at the time the instrument is designated as being in the signal path.

- g. *Public Property Get StaticVSDyn() As Integer***: indicates whether the instrument is currently in a dynamic or static mode. where the returned value is 0 for “dynamic” and 1 for “static”. If designated as “static”, the instrument, during automated measurements, stays in the setup state determined by the user, where the state is set either during schedule editing or prior to an interactive-automated measurement. If designated as “dynamic”, a measurement DLL is given the permission to alter the state of the instrument during execution of the measurement routine. There is no way to prevent the measurement routine from changing the state of the instrument even if it is designated as “static”, and therefore, this option can be overridden if necessary. However, the measurement routine should query for this setting using the Property *Get StaticVsDyn()*, so that a decision can be made as to whether to grant the user the right to have static control of the instrument. Whether the instrument is to be designated as “static” or “dynamic” is determined at the time the user designates the signal path in the system configuration form. The user is automatically given this choice at the time the instrument is designated as being in the signal path.
- h. *Public Property Get DynParam() As Byte()***: returns information about the instrument dynamic configuration - packaged as a a byte array. If the instrument is in *Passive Recorded-instrument-settings mode*, then the parameters returned are simply those which describe the “virtual state” of the instrument. (“Virtual state” is the state of the instrument had the commands actually been sent.) The parameters will be packaged by the *Dynamic Parameters* form, which updates the information about the current instrument state before returning the information. The following five header entries will be located at the beginning of the byte array:

  - (1) A preamble containing the following characters: “RSMS4G\_InstDynConfigPreamble”. The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d) , and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the

end.

- (2) A version number to designate the packaging version. There may be more than one way in which the information is packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
- (3) The name of the Instrument ActiveX file responsible for packaging the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., HP8566.DLL )
- (4) The type of instrument/component, followed by a null character (hex 0) to mark the end of the name (e.g., Fixed Filter). The name should be identical to the string returned by *Function InstInfo* in class *clsInstPubInterface*.
- (5) An indicator which designates whether the instrument is in “dynamic” or “static” mode. The static-vs-dynamic indicator should be a numeric character followed by a null character (hex 0) to mark the end of the string. 0 = “static”, and 1 = “dynamic”.
- (6) The number of bytes in the byte array - including these initial six header entries. The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should be preceded by zeros. (e.g., 0000532).

*i. Public Sub InstDynConfigPackaging(ByRef PathAndFile As String):* opens and appends to an ASCII file, designated by *PathAndFile* (full path and file name), the dynamic component-/instrument-configuration packaging for each instrument/component type. The format of the output must be as follows, where quotes designate a required title, strings inclosed in <> designate the value, and words in italics simply give an explanation and are not part of the output:

I) “INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION PACKAGING” *this is a title for the section.*

A) “FOR:” *this is a subtitle - indented X 1*

1) “Instrument / Component Type: ” <string> *where the sting represents a description of the instrument / component type - indented X 2.*

2) “Dynamic Configuration Packaging Version: ” <value> where **value** represents the version number of the configuration packaging - integers only - indented X 2.

3) “ Date version originated: ” <MM-DD-YY> where **MM**=month, **DD**=day, **YY**=year, and must be represented by two digits for each value. (e.g., Date version originated 01-15-03) - indented X 2.

B) “FORMAT:” this is a subtitle - indented X 1.

1) “INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION HEADER:” this is a subtitle - indented X 2

“ Header var #1 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

“ Header var #2 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

“ Header var #3 ” <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the header variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

2) “INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION PARAMETERS:” this is a subtitle - indented X 2

“Parameter #1: ” <description>; <type>; <# of bytes>;(<units>) - indented X 3

“Parameter #2: ” <description>; <type>; <# of bytes>;(<units>) - indented X 3

“Parameter #3: ” <description>; <type>; <# of bytes>;(<units>) - indented X 3

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

3) “INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION FOOTER:” this is a subtitle - indented X 2

“ Footer var #1 ” <description>; <type>; <# of bytes>;

(<units>) - indented X 3

“ Footer var #2 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

“ Footer var #3 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

---

### EXAMPLE:

#### INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION PACKAGING

##### FOR:

Instrument / Component Type: Spectrum Analyzer

Dynamic Configuration Packaging Version: 1

Date version originated: 04-15-03

##### FORMAT:

#### INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION HEADER:

Header Var #1: Preamble; character bytes

“RSMS4G\_InstDynConfigPreamble”;byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable; (none)

Header Var #2: version number; numerical characters terminated with a null character; variable; (none)

Header Var #3: Instrument ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #4: type of component/instrument; characters terminated with a null character; variable; (none)

Header Var #5: number of data bytes - including header, parameters, and footer; numerical characters terminated with a null character; variable; (none)

#### INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION PARAMETERS:

Parameter #1: Sweep time / Dwell time; integer; 2 bytes; (seconds)

Parameter #2: Center frequency; double; 8 bytes; (MHz)

Parameter #3: Frequency span; double; 8 bytes; (MHz)

Parameter #4: Resolution bandwidth; double; 8 bytes; (MHz)

Parameter #5: Video bandwidth; double; 8 bytes; (MHz)

INSTRUMENT- / COMPONENT-DYNAMIC-CONFIGURATION FOOTER:

None

---

- j. *Public Property Let DynParam(ByRef Val() as Byte)***: receives a byte array containing the information about the instrument dynamic configuration, un-packages the array into meaningful information, and then shows the dynamic parameters via the form *Dynamic Configuration form*. The instrument, whether virtual or real, should also be set up according to the header information that designates “dynamic” vs “static” mode. Since a “version” or “type” marker follows the AcitveX file name, interpretation is backward compatible with older packaging versions. Property *Let DynParam* must be called prior to any call to Property *Let DataDump*.
- k. *Public Sub ParseDynParam(ByRef Val() as Byte)***: receives a byte array containing the information about the instrument dynamic configuration, un-packages the array into meaningful information, but does NOT shows the dynamic parameters via the form *Dynamic Configuration form*. Since a “version” or “type” marker follows the AcitveX file name, interpretation is backward compatible with older packaging versions.
- l. *Public Property Let ConfigByParam(ByRef Val() as Byte)***: receives a byte array containing the information about the instrument dynamic configuration, un-packages the array into meaningful information, and then sets up the physical instrument, as well as the virtual panel, and the dynamic configuration form consistent with the dynamic configuration parameters. How the virtual panel is displayed depends upon the current usage mode. Since a “version” or “type” marker follows the AcitveX file name, interpretation is backward compatible with older packaging versions.
- m. *Public Sub DynParam2ASCII(ByRef ParamVal() as Byte,ByRef PathAndFile As String)***: receives, through argument *ParamVal*, a byte array containing instrument configuration information (originally obtained through *Property Get DynParams* from the object of type *clsInstPubInterface*), un-packages the array into meaningful information (based upon the version number), opens, for appending, the file designated by the argument *FileAndPath*, writes the measurement data along with appropriate labels to an ASCII file, and then closes the file.
- n. *Public Sub Trace2ASCII(ByRef DataBytes() As Byte,ByRef PathAndFile As String)***: receives, through argument *DataBytes*, a byte array containing a trace data (originally obtained through *Property Get DataDump* from the object of type *clsInstPubInterface*), un-packages the byte array to create meaningful information (based upon the version number), opens, for appending, the file designated by the argument *FileAndPath*, writes the measurement data along with appropriate labels to an ASCII file, and then closes the file. This

subroutine will not be called if the data was originally obtained by calling *Property Get DataStream*.

- o. *Function InstErrs(ByRef ErrStr() As String) As integer***: this function requests that a query be made to the instrument in order to determine if there are any instrument errors. It returns, by reference, an array of strings containing any and all instrument errors messages as a result of the query. The function returns an integer indicating the number of errors (0 = no errors).
- p. *Public Property Get ICmmdQueryObj() As Object***: returns, by reference, the instantiated GENERAL INTERFACE to the command/query class.
- q. *Public Property Get CmmdQueryObj() As Object***: returns, by reference, the instantiated command/query class as an object.
- r. *Public Sub UpdateVPTrace()***: Notifies the Virtual Panel to perform a single update to the virtual panel trace with the most recent instrument trace data. Not all instruments will implement this subroutine, as some do not have a trace (e.g. preselectors). This is ignored if the virtual panel is minimized.
- s. *Public Sub UpdateVPParam()***: Notifies the Virtual Panel to update the panels settings consistent with the physical instrument (only those parameters currently being displayed by the virtual panel). NOTE: This does NOT result in update of the parameters within the *Dynamic Configuration* form. This is ignored if the virtual panel is minimized.
- t. *Public Function InstInfo() As String***: returns a string containing the name of the instrument model, and instrument type, separated by a comma. Example: E4440,Spec Analyzer
- u. *Public Property Let VPMode(ByRef MinimizeVal As Integer, ByVal vMode as Integer)***: sets the *usage mode* (described above), where the argument *MinimizeVal* designates whether to minimize the virtual panel (0 = Default display mode, 1 = minimized to the toolbar), and the argument *vMode* designates one of the following, where the value passed may be designated as a enumerated type *VPUsageMode* contained in *Common.base*:

  - (1) 0 = ***Fully manual mode*** (the coder can use “FULL\_MANUAL” of the enumerated type *VPUsageMode*): The Virtual Panel is displayed (if not already displayed), and the user designates the settings and physically changes the instrument settings in real time by sending commands to the device.

    - (a) ***Local /Remote*** - enabled
    - (b) ***Preset*** - enabled
    - (c) ***Save-data*** - enabled: when depressed, this button calls the subroutine *SaveTrace()* in the object of *clsCallBack*, which is then used to raise an event by the form of *frmNonSchedExecutor*. This,

in turn, results in a call to *Property Get DataDump()* of the *clsInstPubInterface* object for the purpose of obtaining the trace data from the instrument DLL.

- (d) **Signal-path** - enabled: when asserted, this option calls the subroutine *SeeSignalPath()* in the object of *clsCallBack*, which is then used to raise an event trapped by the form of *frmNonSchedExecutor*. This, in turn, calls subroutine *SetHrdwrCnfgAndPath()* in the object of type *clsConfigPubInterface* to show and set the focus of the form of type *frmSystemConfig*. If a change is made to the signal path, the object of *clsConfigPubInterface* raises the event *SigPathChanged()* which is then trapped by either form of type *frmIntAutExec*, or *frmFullManualExec*, . The executor then notifies, via subroutine *PathHasChngd()*, any extraneous-measurement DLL, and/or antenna-position-control DLL that the path has changed. Should this occur, each of the DLLs immediately relinquish all instrument Access IDs, each reference to an object of type *clsInstPubInterface* is set to “Nothing”, and then access rights are re-established for all necessary instruments for which control is required.
  - (e) **Save-to-ASCII** - disabled
  - (f) **Re-measure** - disabled
  - (g) **Save/Recall Configuration** - enabled
  - (h) **Print** - enabled
  - (i) **Other instrument control** - enabled
  - (j) **Display** - enabled
- (2) 1 = **Immediate-instrument-setup mode** (the coder can use “IMM\_INST\_SETUP” of the enumerated type *VPUUsageMode*): The Virtual Panel is displayed (if not already displayed), and the user designates the settings and physically changes the instrument settings in real time by sending commands to the device.
- (a) **Local /Remote** - enabled
  - (b) **Preset** - enabled
  - (c) **Save-data** - disabled
  - (d) **Signal-path** - enabled: when asserted, this option calls subroutine *SeeSignalPath()* in the object of type *clsCallBack*. This in turn raises an event in the measurement object of type *clsMeasPubInterface* (during an interactive-automated measurement), which then calls subroutine

*SetHrdwrCnfgAndPath()* in the object of type *clsConfigPubInterface* to show and set the focus of the form of type *frmSystemConfig*. If a change is made to the signal path, the object of *clsConfigPubInterface* raises the event *SigPathChanged()* which is then trapped by either form of type *frmIntAutExec*, or *frmFullManualExec*. The executor then notifies, via subroutine *PathHasChngd()*, any active measurement DLL, extraneous-measurement DLL, and/or antenna-position-control DLL that the path has changed. Should this occur, each of the DLLs immediately relinquish all instrument Access IDs, each reference to an object of type *clsInstPubInterface* is set to “Nothing”, and then access rights are re-established for all necessary instruments for which control is required. If the *signal-path* option is asserted in the virtual panel simply during setup of the system configuration and signal path (no interactive-automated measurement), a call is still made to the subroutine *SeeSignalPath()* in the object of type *clsCallBack*, but since there is no object to trap the raised event, nothing happens.

- (e) ***Save-to-ASCII*** - disabled
  - (f) ***Re-measure*** - disabled
  - (g) ***Save/Recall Configuration*** - enabled
  - (h) ***Print*** - disabled
  - (i) ***Other instrument control*** - enabled
  - (j) ***Display*** - enabled
- (3) 2 = ***Recorded-instrument-setup mode – active*** (the coder can use “REC\_INST\_SETUP\_ACTIVE” of the enumerated type *VPUsageMode*): The Virtual Panel is displayed (if not already displayed), and the user designates the settings and physically changes the instrument settings in real time by sending commands to the device. As the commands are executed, the dynamic configuration of the instrument changes. When Property *Get DynParam* is called, the dynamic configuration of the instrument can be packaged and returned in a byte array.
- (a) ***Local /Remote*** - enabled
  - (b) ***Preset*** - enabled
  - (c) ***Save-data*** - disabled
  - (d) ***Signal-path*** - disabled:
  - (e) ***Save-to-ASCII*** - disabled
  - (f) ***Re-measure*** - disabled

- (g) **Save/Recall Configuration** - enabled
  - (h) **Print** - disabled
  - (i) **Other instrument control** - enabled
  - (j) **Display** - enabled
- (4) 3 = **Recorded-instrument-setup mode – passive** (the coder can use “REC\_INST\_SETUP\_PASSIVE” of the enumerated type VPUsageMode): The Virtual Panel is displayed in *Passive Display* mode (if not already displayed), and the user can record a series of instrument commands without any physical connection to the instrument. As the commands are issued, the virtual state of the instrument is determined and stored so that when Property *Get DynParam* is called, the virtual dynamic configuration of the instrument can be packaged and returned in a byte array. When in this mode, the command/query module will not be used since no physical connection is established.
- (a) **Local / Remote** - disabled
  - (b) **Preset** - enabled but does not send commands to instrument
  - (c) **Save-data** - disabled
  - (d) **Signal-path** - disabled:
  - (e) **Save-to-ASCII** - disabled
  - (f) **Re-measure** - disabled
  - (g) **Save/Recall Configuration** - enabled
  - (h) **Print** - disabled
  - (i) **Other instrument control** - enabled but does not send commands to instrument
  - (j) **Display** - disabled
- (5) 4 = **Instrument-observation mode** (the coder can use “INST\_OBSERVATION” of the enumerated type VPUsageMode): The Virtual Panel is displayed (if not already displayed) so that the display can be observed during automated control of the instrument.
- (a) **Local / Remote** - disabled
  - (b) **Preset** - disabled
  - (c) **Save-data** - disabled
  - (d) **Signal-path** - disabled:
  - (e) **Save-to-ASCII** - disabled
  - (f) **Re-measure** - disabled
  - (g) **Save/Recall Configuration** - disabled

- (h) **Print** - disabled
  - (i) **Other instrument control** - disabled
  - (j) **Display** - enabled
- (6) 5 = **Data-preview mode** (the coder can use “DATA\_PREVIEW” of the enumerated type VPUUsageMode): The Virtual Panel is displayed (if not already displayed) so that the display can be used to examine recorded data acquired from a fully-manual measurement. When in this mode, the command/query module will not be used since no physical connection is established.
- (a) **Local / Remote** - disabled
  - (b) **Preset** - disabled
  - (c) **Save-data** - disabled
  - (d) **Signal-path** -enabled: when asserted, this option calls the subroutine *SeeSignalPath()* in the object of *clsCallBack*, which is then used to raise an event trapped by the form of type *frmNonSchedExecutor* (for data acquired during a *fully-manual* measurement). This, in turn, calls subroutine *SetHrdwrCnfgAndPath()* in the object of type *clsConfigPubInterface* to show and set the focus of the form of type *frmSystemConfig*. When in *data-preview* mode, the System-Configuration-Form is also set in the same mode, and therefore, the system configuration and signal path cannot be changed.
  - (e) **Save-to-ASCII** - enabled
  - (f) **Re-measure** - enabled
  - (g) **Save/Recall Configuration** - disabled
  - (h) **Print** - enabled
  - (i) **Other instrument control** - disabled
  - (j) **Display** - enabled
- (7) 6 = **Setup-preview mode** (the coder can use “SETUP\_PREVIEW” of the enumerated type VPUUsageMode): In this mode, the virtual panel is hidden and only the form of type *frmDynamicConfiguration* is available to the user, where the default is to minimize this form. When in this mode, the command/query module will not be used since no physical connection is established.
- (8) 6 = **stealth mode** (the coder can use “STEALTH” of the enumerated type VPUUsageMode): In this mode, forms are instantiated but none are shown. This mode is primarily used for the purpose of parsing the data from the data record and making it available for processing without any

display of the data. The argument *MinimizeVal* is ignored. When in this mode, the command/query module will not be used since no physical connection is established.

v. **Public Property Get VPMODE(ByRef MinimizeVal As Integer) As integer:** Returns the usage mode designated one of the following:

- (1) 1 = *Fully manual mode:*
- (2) 2 = *Immediate-instrument-setup mode:*
- (3) 3 = *Recorded-instrument-setup mode – active:*
- (4) 4 = *Recorded-instrument-setup mode – passive:*
- (5) 5 = *Instrument-observation mode :*
- (6) 6 = *Data-preview mode:*
- (7) 7 = *Setup-preview mode:*
- (8) 8 = *Stealth mode:*

The argument *MinimizeVal* is passed back by reference and designates whether the virtual panel is minimized (0 = Default display mode, 1 = minimized to the toolbar)..

w. **Public Property Let VPVisibility(ByVal VisVal as Integer):** sets the visibility of the virtual panel, where 0 = hide (remains instantiated), 1 = visible. In *stealth mode*, the virtual panel remains hidden.

x. **Public Property Get VPVisibility() as Integer:** returns an integer designating whether the virtual panel is currently visible or not. 0 = not visible, 1 = visible.

y. **Public Property Get DynCnfgMD5() As Byte():** returns the 16 byte array containing the MD5 Hash Code for the current instrument dynamic configuration - generated from the byte array (excluding header information), same as that returned by *Property Get DynParam* of *clsInstPubInterface*.

z. **Public Property Let DataDump(ByRef DataBytes() As Byte):** receives a byte array containing data and passes the *DataBytes* argument as a byte array (created by *Property Get DataDump*), un-packages the byte array to create meaningful information (based on the version number) and displays the data on the *Virtual Panel Form* when in *Data-preview mode*. *Property Let DynParam* must be called prior to *Property Let DataDump* and the virtual panel should show settings consistent with the values passed *Property Let DynParam*. This property can be called by the executor only if by calling *Get PkgType* in the object of type *clsInstPubInterface* the returned value is "0" or if the record header information indicates that the data is packaged as a byte array (as apposed to being streamed into a temporary holding file).

aa. **Public Sub ParseDataDump(ByRef DataBytes() As Byte):** receives a byte

array containing data and passes the *DataBytes* argument as a byte array (created by Property *Get DataDump*), un-packages the byte array to create meaningful information (based on the version number) but does NOT display the data on the *Virtual Panel Form* when in *Data-preview mode*. This is used primarily for parsing a record from a data file for the purpose of processing without displaying the data. This property can be called by the executor only if by calling *Get PkgType* in the object of type *clsInstPubInterface* the returned value is "0" or if the record header information indicates that the data is packaged as a byte array (as apposed to being streamed into a temporary holding file).

- bb. Public Property *Get PkgType* () As Integer:** returns an integer which indicates the method by which data is to be passed. When *PkgType* = 0, the data is packaged as byte array and the executor should call Property *Get DataDump* in the object of type *clsInstPubInterface* to retrieve the packaged data and, in turn, call Sub *WriteDataRecord* to pass the data on to the *File I/O Manager*. When *PkgType* = 1, the data, including header, is contained in a temporary holding file in which large quantities of data were streamed. In this latter case, the executor should retrieve the file name by calling Property *Get DataStream* in the object of type *clsInstPubInterface* and then pass it on to the *File I/O Manager* by calling Sub *WriteDataStream*.
- cc. Public Property *Get DataDump*() As Byte():** This property should be called by the executor only after calling Property *Get PkgType* in the object of type *clsInstPubInterface* and receiving a return value of "0". The implementation of this property is mutually exclusive to *Property Get MeasStream*. In other words, only one of the two properties "*Get DataStream*" and "*Get DataDump*" is implemented in agreement with Property *Get PkgType*. The other shows an error if called. The property returns, from a virtual panel, data packaged as a byte array with the following four header entries located at the beginning of the array:
- (1) A preamble containing the following characters: "RSMS4G\_DataPreamble". The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d) , and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the end.
  - (2) A version number to designate the version of the data packaging. There may be more than one way in which the data are packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
  - (3) The name of the Instrument ActiveX file responsible for packaging the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., HP8566.DLL )

- (4) The number of bytes in the data package (including these four header entries). The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should be preceded by zeros. (e.g., 0000532).

At the end of the Data (data footer) is a 32 Bit CRC Code for the data only (to identify data corruption). Code for the CRC32 algorithm is located in Common.bas. The code takes an array of bytes as input and computes a 32-bit "checksum". To use the algorithm, call InitialiseCRC32tab and then call GetCRC32ForByteArray(Bytes).

**dd. *Public Property Get DataStream() As Byte()*:** This property should be called by the executor only after calling Property *Get PkgType* in the object of type *clsMeasPubInterface* and receiving a return value of "1". The implementation of this property is mutually exclusive to *Property Get DataDump*. In other words, only one of the two properties "*Get DataStream*" and "*Get DataDump*" is implemented in agreement with Property *Get PkgType*. The other shows an error if called. This property returns, from a virtual panel, a temporary holding file containing the data dump preceded by the following four header entries located at the beginning of the file:

- (1) A preamble containing the following characters: "RSMS4G\_DataPreamble". The preamble is preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) to mark the beginning. In addition, the preamble is followed by a null character (hex 0) to mark the end.
- (2) A version number to designate the version of the data packaging. There may be more than one way in which the data are packaged into a byte array. Each method is associated with a version. The version number is represented by numeric characters followed by a null character (hex 0) to mark the end of the string.
- (3) The name of the Instrument ActiveX file responsible for packaging the information, followed by a null character (hex 0) to mark the end of the file name. (e.g., HP8566.DLL )
- (4) The number of bytes in the data package (including these four header entries). The number of bytes is represented by numeric characters followed by a null character (hex 0) to mark the end of the string. A

variable length entry could result in a recursive situation, whereby the actual byte length is changed as this entry is added to the header. Therefore, this entry should consist of a fixed number of numeric characters, the length of which exceeds the number of digits required to represent the maximum possible size of the byte array that could be passed back during the call to this property. Significant figures should be preceded by zeros. (e.g., 0000532).

At the end of the Data (data footer) is a 32 Bit CRC Code for the data only (to identify data corruption). Code for the CRC32 algorithm is located in Common.bas. The code takes an array of bytes as input and computes a 32-bit "checksum". To use the algorithm, call InitialiseCRC32tab and then call GetCRC32ForByteArray(Bytes).

ee. **Public Sub InstDataDumpPackaging(ByRef PathAndFile As String)** opens and appends to an ASCII file, designated by *PathAndFile* (full path and file name), the data packaging for each version of the particular instrument data-dump (whether the data is passed as a byte array or streamed into a temporary holding file). This refers to data that has been gathered by a virtual panel during a fully manual measurement, whereby the user chooses to save the trace(s) to a data file. The format of the output must be as follows, where quotes designate a required title, strings inclosed in <> designate the value, and words in italics simply give an explanation and are not part of the output:

I) "DATA-DUMP PACKAGING" *this is a title for the section.*

A) " FOR:" *this is a subtitle - indented X 1.*

1) " Instrument Model: " <string> *where the sting represents a description of the instrument model - indented X 2.* This string should be identical to that received via *Function InstInfo* of class *clsInstPubInterface*.

2) " Data-dump Packaging Version: " <value> *where "value" represents the version number of the measurement-data packaging - integers only - indented X 2*

3) " Date version originated: " <MM-DD-YY> *where MM= month, DD=day, YY=year, and must be represented by two digits for each value. (e.g., Date version originated 01-15-03) - indented X 2.*

B) "FORMAT:" *this is a subtitle - indented X 1*

1) "DATA HEADER:" *this is a subtitle - indented X 2*

" Header var #1 " <description>; <type>; <# of bytes>;  
(<units>) - indented X 3

“ Header var #2 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

“ Header var #3 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the header variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

2) “DATA:” *this is a subtitle - indented X 2*

a) “Number of Variables: ” <value> where **value** represents the number of independent and dependent data variables - *indented X 3.*

b) *Description of the variables:*

“ Variable #1 ” <description>; <type>; <# of bytes>; ,  
(<units>) - *indented X 3*

“ Variable #2 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

“ Variable #3 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

c) “ Order ” <sequential | alternated> where the order is either **sequential** (all of the values for variable #1 written first, then all the values for variable #2, etc ) or **alternated** (first value of all variables written first, followed by second value of all variables, etc.) - *indented X 3*

3) “DATA FOOTER:” *this is a subtitle - indented X 2*

“ Footer var #1 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

“ Footer var #2 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

“ Footer var #3 ” <description>; <type>; <# of bytes>;  
(<units>) - *indented X 3*

*etc., where **type** is a string describing the Visual Basic*

*variable type (e.g. integer, long, double, etc), # of bytes designates the number of bytes in the footer variable, **description** is a string describing what the variable represents, and **units** is a string describing the units.*

---

**EXAMPLE:**

MEASUREMENT-DATA PACKAGING

FOR:

Instrument Model: HP9566

Data-dump Packaging Version: 1

Date version originated: 01-23-03

FORMAT:

DATA HEADER:

Header Var #1: Preamble; character bytes "RSMS4G\_DataPreamble";byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable; (none)

Header Var #2: version number; numerical characters terminated with a null character; variable; (none)

Header Var #2: Instrument ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #3: number of data bytes - including data header, data, and data footer; numerical characters terminated with a null character; variable;(none)

DATA:

Number of Variables: 3

Variable #1: Frequency; double; 8 bytes; (MHz)

Variable #2: Trace A Magnitude; double; 8 bytes; (dBm)

Variable #3: Trace B Magnitude; double; 8 bytes; (dBm)

Order: alternated

DATA FOOTER

Footer var #1: CRC of data; byte array; 4;(none)

MEASUREMENT-DATA PACKAGING

FOR:

Instrument Model: HP9566

Data-dump PackagingVersion: 2

Date version originated: 06-10-03

FORMAT:

DATA HEADER:

Header Var #1: Preamble; character array “RSMS4G\_DataPreamble”; byte array preceded by a carriage return (hex 0d), a linefeed (hex 0a), another carriage return (hex 0d), and another linefeed (hex 0a) and terminated with a null character (hex 00); variable; (none)

Header Var #2: version number; numerical characters terminated with a null character; variable;(none)

Header Var #2: Instrument ActiveX file responsible for packaging the information; characters terminated with a null character; variable; (none)

Header Var #3: number of data bytes - including data header, data, and data footer; numerical characters terminated with a null character; variable;(none)

DATA:

Number of Variables: 2

Variable #1: Frequency; double; 8 bytes; (MHz)

Variable #2: Magnitude; double; 8 bytes; (dBm)

Order: sequential

DATA FOOTER

Footer #1: CRC of data; byte array; 4;(none)

---

**WE SHOULD ADD SOMETHING LIKE THE FOLLOWING FOR GETTING THE INSTRUMENT SETUP PARAMETERS**

*ff. Public Function GetCoreData(ByRef FrmtColmns as Integer, ByRef FrmtRows as Long, ByRef FrmtMode as Integer, ByRef FrmtVarType() as Integer, ByRef FrmtVarLabels() as VarLabels, ByRef FrmVarUnits() As VarUnits) As Byte():* This function returns an array of bytes containing only the data - without a header - the purpose being to provide un-packaged data for data processing routines. The data will represent the actual value and will not require scaling and/or offset. The data will also be raw, meaning that it is not corrected by any calibration factor. All independent and dependent variables will have the same vector length, so that, for every value in the independent variable, there are corresponding values in each dependent variable. The array of bytes is formatted in a manner described by the variables in the argument list. These are as follows:

- (1) *FrmtColmns*: This is an integer returned by reference that designates the number of data variables (columns).
- (2) *FrmtRows*: This is a long integer returned by reference that designates