SANDIA REPORT

SAND2019-6653

Printed June 2019

Sandia National Laboratories

Application Note: Mixed Signal Simulation with Xyce™ 6.11

Peter E. Sholander, Richard L. Schiek

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

This application note describes how Release 6.11 of the Xyce circuit simulator can be coupled with external simulators via either a Python-based interface that leverages the Python ctypes foreign function library or via the Verilog Procedural Interface (VPI). It also documents the usage of these interfaces on RHEL6 and RHEL7, with Python 2.6 or 2.7. These interfaces are still under development and may change in the future. So, a key purpose of this application note is to solicit feedback on these interfaces from both internal Sandia Xyce users and other performers on the DARPA Posh Open Source Hardware (POSH) program.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Xyce is Sandia National Laboratories' SPICE-compatible high-performance analog circuit simulator, written to support the simulation needs of the laboratories' electrical designers. It has the capability to solve extremely large circuit problems on large-scale parallel computing platforms, and contains device models specifically tailored to meet Sandia's needs.

This application note documents recent work on interfacing Release 6.11 of Xyce to both Verilog and VHDL (VHSIC Hardware Description Language) simulation codes. These interfaces are still under development and may change in the future. So, a key purpose of this application note is to solicit feedback on these interfaces from both internal Sandia Xyce users and other performers on the DARPA Posh Open Source Hardware (POSH) program.

Chapter 2 gives a description of the `XyceCInterface` class and its methods. It also describes how to build Xyce as "shared" objects that can be invoked by, or linked with, other programs. That `XyceCInterface` class provides the basis of the Python-based and VPI-based interfaces that are the described in Chapters 3 and 4. Working examples are given for both of these interfaces. Finally, this application note only documents the usage of these interfaces on RHEL6 and RHEL7, with Python 2.6 or 2.7. Their support on OSX and Windows, as well as compatibility with Python 3, is "future work".

Reference [1] describes the Xyce General External Interface, which is another mechanism for external simulation codes to use Xyce as their circuit simulator. That approach can be used on a wide variety of circuit/mesh coupling problems. An example is coupling frequency-domain electromagnetic simulators to Xyce, and performing the frequency-domain analyses that Xyce provides such as harmonic balance.

1.1. TARGET AUDIENCE AND PREREQUISITES

This application note is intended for users and developers of existing simulation codes who wish to use Xyce in order to add circuit simulation capability to their existing capabilities. It assumes that you have already downloaded and compiled **Xyce**™ according to its documentation, that you have installed it in a manner that allows you to run it directly by typing "Xyce" in the command line, and that you are able to run a basic netlist using that installed copy of Xyce. Section 2.2 then gives more instructions of how to compile and install Xyce as "shared objects" that can be linked with the open-source Verilog simulator Icarus [2] via the Verilog Procedural Interface (VPI) [3], or invoked via the Sandia-supplied Python interface.

For external open-source users, source code for Xyce can be obtained from our website at xyce.sandia.gov. Internal Sandia users should contact the Xyce development team for either source code access or access to a build of the shared-objects version of Xyce. That capability is not included, by default, in the Xyce 6.11 binaries that are distributed within Sandia.

The Xyce Reference Guide [4] and Users' Guide [5] provide more detail on Xyce syntax and usage for circuit simulation. Readers who are not familiar with SPICE or Xyce are encouraged to work through the tutorial examples in Chapters 2 and 3 of the Users' Guide before trying to run the examples given in this application note. Those two chapters explain how to run transient (.TRAN) simulations in Xyce, using a simple Diode Clipper circuit as an example. Reference [6] contains a brief explanation of the mathematical foundations of parallel circuit simulation in Xyce. All of these documents are available on our website at xyce.sandia.gov.

This application note assumes minimal familiarity with Verilog. So, section 4.1 gives a brief overview of Icarus, which is an open-source Verilog simulation and synthesis tool.

One purpose of this application note is to solicit feedback on these Mixed Signal Interfaces. The Xyce development team can be contacted via email at xyce@sandia.gov.

2. XYCECINTERFACE

The XyceCInterface class provides methods to invoke various methods on a pointer to an N_CIR_Xyce object (whose class name is Xyce::Circuit::Simulator), which is the topmost object in a Xyce simulation. Section 2.1 provides a detailed description of the methods provided the XyceCInterface class. The parameters, return values, known limitations and bugs for each method are described. Examples of how to use these methods are given in subsequent chapters of this application note. Section 2.2 then gives a description of how to build the Xyce 6.11 source code so that it includes the XyceCInterface class and can be linked to, or invoked as, "shared objects" by other programs.

Chapters 3 and 4 describe Python-based and VPI-based interfaces that leverage the XyceCInterface class. However, that class can also be leveraged directly by C++ codes that do not need the full generality of the Xyce General External Interface [1].

2.1. API DESCRIPTION

For the Xyce 6.11 release, the XyceCInterface.C and XyceCInterface.h files are located in the utils/XyceCInterface subdirectory of the Xyce source tree. The names, signatures and return types of these methods may change in future Xyce releases. In addition, slightly different versions and additional methods may be developed for the Python-based and VPI-based interfaces described in subsequent chapters.

2.1.1. xyce_open

```
void xyce_open(void ** ptr)
```

This method allows the calling program to obtain a void** pointer to an N_CIR_Xyce object. It must be called before any of the other methods described below. The type of this pointer may change in future Xyce releases.

2.1.2. xyce_initialize

```
int xyce_initialize(void ** ptr, int argc, char ** argv)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method. The other two arguments for the `xyce_initialize` method mimic the function of the same arguments in a normal C or C++ main function: they are interpreted as representing the command line that invoked Xyce. The argument `argc` is the number of strings present in the array of strings, `argv`.

The string `argv[0]` is taken to be the name of the program, and no use is made of it. Subsequent elements of the `argv` array are command line options as documented in Chapter 3 of the Xyce Reference Guide [4]. The final argument string in this array should be the name of the Xyce netlist to be processed.

The `xyce_initalize` method actually invokes the `initializeEarly` and `initializeLate` methods of the underlying `N_CIR_Xyce` object. The `initializeEarly` method instantiates the devices present in the netlist and allocates all of the solvers and packages needed. The `initializeLate` method then completes the analysis of the circuit topology, sets up the internal vector and matrix storage, initializes the output manager, and makes the `N_CIR_Xyce` object ready for the simulation to take place. If the external programs using the Python-based and VPI-based interfaces described in this application note needed to set Xyce-internal device properties directly, rather than via the simulation's Xyce netlist, then the existing `xyce_initalize` method of the `XyceCInterface` class could likely be split into separate `xyce_initializeEarly` and `xyce_initialiazeLate` methods. That split approach was taken for the Xyce General External Interface [1].

This method returns a integer value that maps to the `Xyce::Circuit::Simulator::RunStatus` enum values. So this function returns 0 for the run status of "ERROR", 1 for the run status of "SUCCESS" and 2 for the run status of "DONE". More details on these run-status codes are:

"ERROR" signifies failure of the initialization, and the actual error condition will have been printed to Xyce's standard error stream. Further calls to that XyceCInterface object's methods should not be made, as Xyce has effectively terminated with a fatal error when this value is returned.

"DONE" signifies that all processing is complete. This return value is used when the command line arguments include an argument that prevents Xyce from proceeding to a full simulation, such as "-syntax", "-count", "-v", "-norun" and so forth. If `xyce_initialize` returns this value, Xyce has effectively exited successfully and further calls such as `xyce_runSimulation` should not be performed.

"SUCCESS" signifies that the initialization was successful, and the XyceCInterface object is ready for futher calls such as `xyce_runSimulation`.

2.1.3. xyce_runSimulation

```
int xyce_runSimulation(void ** ptr)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

This method causes Xyce to run the entire simulation specified in the netlist to completion. It returns the status codes described in the `xyce_initialize` subsection above.

2.1.4. xyce_simulateUntil

```
int xyce_simulateUntil(void **ptr,
                  double requestedUntilTime,
                  double * completedUntilTime)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

This method causes Xyce to perform a limited simulation not to exceed the simulation time specified in `requestedUntilTime`. Upon return, `completedUntilTime` will contain the actual time that Xyce reached, which will be less than or equal to `requestedUntilTime` either because the netlist specified a final time earlier than `requestedUntilTime`, or because there was a fatal convergence error. Each call to `xyce_simulateUntil` after the first one resumes the current simulation from where the last call left off. If `xyce_simulateUntil()` is called with `requestedUntilTime` less than the current simulation time then the simulation will proceed to completion from that current simulation time.

This method returns 1 if the simulation completed successfully, either by reaching the value of `requestedUntilTime` or the final time specified in the netlist, whichever is earlier. It returns 0 if the run was unsuccessful. If `xyce_simulateUntil` returns 1 and `completedUntilTime` is less than `requestedUntilTime` then Xyce has completed its work and further calls to `xyce_simulateUntil` will do nothing.

2.1.5. xyce_close

```
void xyce_close(void ** ptr)
```

This method causes Xyce to close all output files after a simulation run is complete and emit timing information. It also deletes the pointer to the `N_CIR_Xyce` object. It should be called after the Xyce simulation is complete.

2.1.6. xyce_getNumDevices

```
int xyce_getNumDevices(void **ptr,
                       char * modelGroupName,
                       int * numDevNames,
                       int * maxDevNameLength)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_getNumDevices` takes a character array containing a "model group" name, and returns the number of devices from the model group in the netlist. It also return the size of longest device name from that model group. It is a general purpose method that can be given any valid model group name ("M" for MOSFETs, "Q" for BJTs, etc. [4]).

This method was added to improve memory management for the `xyce_getDeviceNames`, `xyce_getDACDeviceNames` and `xyce_getADCMap` methods. For C interfaces, it allows the calling program to pre-allocate the correct-sized arrays for those methods' returned parameter(s) such as `deviceNames`. It is also used internally for array-size management by the corresponding Python methods such as `getDeviceNames`.

This method returns 1 if at least one device of the requested type exists in the netlist. Otherwise, it returns 0 with both `numDevNames` and `maxDevNameLength` equal to 0. A request for an invalid model group will also return 0.

2.1.7. xyce_getDeviceNames

```
int xyce_getDeviceNames(void ** ptr,
                        char * modelGroupName,
                        int * numDevNames,
                        char ** deviceNames)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_getDeviceNames` takes a character array containing a "model group" name, and returns a char** array of the names for all devices in the netlist of that type. It is a general purpose method that can be given any valid model group name ("M" for MOSFETs, "Q" for BJTs, etc. [4]). This method also returns the number of devices from the specified model group in the netlist.

This method returns 1 if at least one device of the requested type exists in the netlist. Otherwise, it returns 0 with `numDevNames` equal 0 and `deviceNames` being of zero length. A request for an invalid model group will also return 0.

2.1.8. xyce_getDACDeviceNames

```
int xyce_getDACDeviceNames(void ** ptr,
                           int * numDevNames,
                           char ** deviceNames)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method.

`xyce_getDACDeviceNames` returns a char** array of the names for all the Digital-to-Analog (DAC) devices in the netlist. So, it is basically a specialized version of the more general `xyce_getDeviceNames` method described above. This method also returns the number of DAC devices in the netlist.

This method returns 1 if at least one DAC device exists in the netlist. Otherwise, it returns 0 with `numDevNames` equal 0 and `deviceNames` being of zero length.

2.1.9. xyce_updateTimeVoltagePairs

```
int xyce_updateTimeVoltagePairs(void ** ptr,
                                char * DACname,
                                int numPoints,
                                double * timeArray,
                                double * voltageArray)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`. If `DACname` is not the name of a valid DAC device in the Xyce netlist then the function will execute with a Xyce warning message and return 0 as noted below.

This method will return 1 if the time-voltage pairs for the specified `DACname` were successfully updated. Otherwise, it will return 0.

The "error condition" of `timeArray` and `voltageArray` being of unequal lengths is checked when this method is invoked via the Python ctypes-based interface. It is not checked when `xyce_updateTimeVoltagePairs` is invoked directly. For direct invocation, it is the responsbility of the calling function to verify that the parameters `numPoints`, `timeArray` and `voltageArray` have consistent values and lengths.

Examples of how to use this method, with both Python and VPI, are provided in the release src subdirectories `utils/XyceCInterface/Python_examples/runCircuitWithDACs` and `utils/XyceCInterface/VPI_examples/runXyceWithDAC`.

2.1.10. xyce_checkResponseVar

```
int xyce_checkResponseVar(void ** ptr, char * variable_name)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_checkResponseVar` takes a character array containing a "measure name". It returns 1 if `variable_name` is a valid measure name in the the Xyce simulation. Otherwise, it returns 0.

An example Xyce measure statement is as follows [4]. This example is a `MAX` measure for a transient (`TRAN`) simulation. Its name is `MAXV1`, where that name is not case-sensitive. It returns the maximum value of the quantity `V(1)` found during the simulation.

```
.MEASURE TRAN MAXV1 MAX V(1)
```

2.1.11. xyce_obtainResponse

```
xyce_obtainResponse(void ** ptr, char * variable_name, double * value)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_obtainResponse` takes a character array containing a "measure" name. It returns the value of that `.MEASURE` statement at the current simulation time in the `value` parameter. If the Xyce simulation has completed then it will return the value at the final simulation time.

This method returns 1 if the requested `variable_name` is a valid measure name in the the Xyce simulation. Otherwise, it returns 0. For a return value of 0, the `value` parameter will also be set to 0.

2.1.12. xyce_setADCWidths

```
int xyce_setADCWidths(void ** ptr,
                      int numADCnames,
                      char ** ADCnames,
                      int *  widths)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_setADCWidths` takes a char** array of the ADC names for which the "output bit-vector widths" are being setting. The parameter `widths` is then an int* array of those widths. Each ADC will then have 2**`width` quantization levels, where different ADCs may have different widths.

This method will return 1 if the "output bit-vector width" is succesfully updated at every ADC specified in `ADCnames`. It will return 0 if the update process fails at any ADC specified in `ADCnames`.

The "error condition" of `ADCnames` and `widths` being of unequal lengths is checked when this method is invoked via the Python ctypes-based interface. It is not checked when `xyce_setADCWidths` is invoked directly. For direct invocation, it is the responsbility of the calling function to verify that the parameters `numADCnames`, `ADCnames` and `widths` have consistent values and lengths.

The ADC widths can be set via this function, the `WIDTH` instance parameter for each individual YADC device, and the associated YADC model parameters (see Section 5.1). The order of precedence is in that order. This function should have the highest precedence, since it occurs after the `xyce_initialize` method is called.

2.1.13. xyce_getADCWidths

```
int xyce_getADCWidths(void ** ptr,
                      int numADCnames,
                      char ** ADCnames,
                      int *  widths)
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_getADCWidths` takes a char** array of the ADC names for which the values of the "output bit-vector widths" are being requested. The parameter `widths` is then an int* array of those widths. Each ADC then has 2**`width` quantization levels, where different ADCs may have different widths.

This method will return 1 if the "output bit-vector width" is succesfully found at every ADC specified in `ADCnames`. It will return 0 if the get process fails for any ADC specified in `ADCnames`. The width value for any ADC not found in the netlist will be returned as 0.

The "error condition" of `numADCnames` and `ADCnames` not being consistent is not checked when `xyce_getADCWidths` is invoked directly. For direct invocation, it is the responsbility of the calling function to verify that the parameters `numADCnames` and `ADCnames` have a consistent value and length.

2.1.14. xyce_getTimeVoltagePairsADC

```
int xyce_getTimeVoltagePairsADC(void** ptr,
                                int * numADCnames,
                                char ** ADCnames,
                                int * numPoints,
                                double ** timeArray,
                                double ** voltageArray )
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_getTimeVoltagePairsADC` returns a char** array of the names for all the ADC devices in the netlist. The returned double** `voltageArray` then contains the "deltaV" values (the differences between the voltages at the positive and negative terminals) for the ADC devices listed in the `ADCnames` array at the time points specified in the double** `timeArray`.

The formats of the returned `numPoints`, `timeArray` and `voltageArray` parameters will be illustrated further in Section 7.1. This function is the "least mature" of the `XyceCInterface` methods and Section 7.1 describes its limitations via a Python-based example. Many of these limitations stem from known limitations in the YADC device (see Section 5.1) implemented in Xyce 6.11.

This method will return 1 if there are ADC devices in the netlist. It will return 0 otherwise.

The use of both `xyce_getTimeVoltagePairsADC` and and `xyce_getTimeStatePairsADC` in the same program is not recommended, because of an open bug.

2.1.15. xyce_getTimeStatePairsADC

```
int xyce_getTimeStatePairsADC(void** ptr,
                              int * numADCnames,
                              char ** ADCnames,
                              int * numPoints,
                              double ** timeArray,
                              double ** stateArray )
```

This method assumes that the pointer `ptr` was previously obtained with the `xyce_open` method and successfully initialized with the `xyce_initialize` method. So, it must be called after the calls to `xyce_open` and `xyce_initialize`.

`xyce_getTimeStatePairsADC` returns a char** array of the names for all the ADC devices in the netlist. The returned double** `stateArray` then contains the integer-valued states (ranging from 0 to $2 * width[i] - 1$, where $width[i]$ is the bit width of the ith ADC) at the time points

specified in the double** timeArray. So, this function partially removes the need for an external simulation program to convert the "deltaV" values, stored internally within Xyce, into state values. However, that external simulator may still need to convert the returned integer state-values into a suitable binary/octal/hexadecimal format for its use.

This method will return 1 if there are ADC devices in the netlist. It will return 0 otherwise.

The use of both xyce_getTimeVoltagePairsADC and and xyce_getTimeStatePairsADC in the same program is not recommended, because of an open bug.

2.1.16. xyce_getADCMap

```
int xyce_getADCMap(void ** ptr,
                   int * numADCnames,
                   char ** ADCnames,
                   int *  widths,
                   double * resistances,
                   double * upperVLimits,
                   double * lowerVLimits,
                   double * settlingTimes)
```

This method assumes that the pointer ptr was previously obtained with the xyce_open method and successfully initialized with the xyce_initialize method. So, it must be called after the calls to xyce_open and xyce_initialize.

xyce_getADCMap returns a char** array of the names of all the ADC devices in the netlist. The widths, resistances, upperVLimits, lowerVLimits and settlingTimes arrays are then int* or double* arrays of the corresponding instance parameters for the ADC devices listed listed in the ADCnames array. The ordering in those five instance-parameter arrays is the same as in the ADCnames array.

This method will return 1 if there are ADC devices in the netlist. It will return 0 otherwise, and the various arrays will be empty in that case.

2.2. XYCE SHARED OBJECTS BUILDING AND TESTING GUIDE FOR RHEL6 AND RHEL7

This section describes how to build the source code for the Xyce 6.11 release as "shared objects" that can be linked with, or invoked by, other simulators. It covers the build process for the gcc compilier on RHEL6 and RHEL7 for a serial build. For information on how to build with the Intel compilers, or on other Linux variants, please contact the Xyce development team.

At this point, a build process for the XyceCInterface code in support of Mixed Signal interfaces is not supported on either OSX or Windows. Support for those operating systems is expected in

future releases though. Finally, the mixed signal interfaces have only been demonstrated with a serial build of Xyce.

The `reconfigure` scripts shown in Figures 2-1 and 2-2 have been shown to work on RHEL6 and RHEL7. They will produce `.so` libraries that can invoked by Python via the Sandia-supplied ctypes interface described in Chapter 3 and also linked with Icarus to create Verilog `vvp` programs as described in Chapter 4. The directory where this build of Xyce will be installed is denoted as `$installDir` in these `reconfigure` scripts. These scripts also refer to the top-level installation directory of Trilinos as `$archdir`, and assume that Trilinos has been built according to the guidance in the Xyce Building Guide [7]. The top-level Xyce src directory is referred to as `$xyceSrcDir`.

```
$xyceSrcDir/configure \
ARCHDIR=$archdir \
--disable-verbose_linear \
--disable-verbose_nonlinear \
--disable-verbose_time \
--enable-shared \
--enable-xyce-shareable \
--prefix=$installDir \
CC=gcc \
CXX=g++ \
F77=gfortran \
CXXFLAGS="-O1 -fno-inline -g"
```

Figure 2-1. Compiling Xyce as Shared Objects on RHEL6 with gcc

```
$xyceSrcDir/configure \
ARCHDIR=$archdir \
--disable-verbose_linear \
--disable-verbose_nonlinear \
--disable-verbose_time \
--enable-shared \
--enable-xyce-shareable \
--prefix=$installDir \
CC=gcc \
CXX=g++ \
F77=gfortran \
CXXFLAGS="-O1 -fno-inline -std=c++11"
```

Figure 2-2. Compiling Xyce as Shared Objects on RHEL7 with gcc

As other notes, the use of `-enable-shared` and `-enable-xyce-shareable` is needed in order to create the `.so` files. Also, the `CXXFLAGS` shown above are set for a debug build. That is

convenient for co-development with Icarus and Xyce, since the combined `vvp` programs (see Chapter 4) can then be debugged in gcc (or your other favorite debugger). A more typical Xyce build would use -O3 instead, for better performance. After running `reconfigure` and `make`, it is recommended that a `make install` also be done.

To test the installed build the following tags list (`-taglist` option for the `run_xyce_regression` script) should be used. It includes the MIXED_SIGNAL tests that are specific to this application note. Per the "Some tests only work when tested from a build directory" section of the "Running the Xyce Regression Suite" web-page [8] the tag `-library` should be used if `make install` was used. The addition of `+mixedsignal` to this tagslist will just run the MIXED_SIGNAL tests.

```
"+serial+nightly-library?noverbose?klu-verbose?rad?fft?qaspr?nonfree?mixedsignal"
```

The MIXED_SIGNAL regression tests have not been fully integrated with the release testing process for Xyce yet. So, they may still be "fragile", especially with respect to how they determine the path to `$xyceSrcDir`. In addition, the Python interface, described in Chapter 3, was tested with Python 2.6.6, 2.7.4 and 2.7.5 for the Xyce 6.11 release. Some features (and regression tests) are known to fail when the tests are run with Python 3.4.2 or 3.5.2. (Note: the regression tests may be hard-coded to use Python 2.x on Sandia systems.) So, the reader should contact the Xyce development team if they have problems with either the build or test processes described in this section.

Finally, this application note only discusses the build process for RHEL6 and RHEL7. The authors do welcome feedback though on the reader's experience with other Linux variants, especially Ubuntu.

2.2.1. Post-Release Code Fixes

If Xyce was built from source then please contact the Xyce Development Team about any relevant updates related to the Mixed Signal Interface. Those updates, if any, should be publically available from our GitHub site (https://github.com/Xyce/).

3. PYTHON WRAPPERS TO XYCECINTERFACE

A Sandia-supplied implementation of ctype-based Python wrappers for the XyceCInterface class is available in the release subdirectory utils/XyceCInterface. The file name is xyce_interface.py. As background, ctypes is a "foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python." More information on ctypes can be found at [9].

This application note will not discuss the internals of the xyce_interface.py file. The key point is that it provides a wrapper for the methods documented in Section 2.1.

3.1. API DESCRIPTION

This section provides a mapping of the Python interfaces methods to the underlying XyceCInterface methods described in Section 2.1.

3.1.1. xyce_interface

```
xyceObj = xyce_interface()
```

This method allows the calling Python program to invoke the underlying xyce_open method of the XyceCInterface. It creates a pointer to an N_CIR_Xyce object (which is also called a "xyce" object in some of the example files discussed in Section 3.2). It must be called before any of the other Python-based methods described below.

3.1.2. initialize

```
result = xyceObj.initialize(argv)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method. The argument (argv) represents the command line that invoked Xyce, but it should not include the program name Xyce. This method allows the calling Python program to invoke the underlying xyce_initialize method of the XyceCInterface. The return value (in result) is the same as for the underlying xyce_initialize method of XyceCInterface.

If initialize() returns 2 (which is the Xyce::Circuit::Simulator::RunStatus of "DONE") then the calling .py file will likely segfault. This can happen for Xyce command line options such as -norun that prevent Xyce from proceeding to a full simulation. (This is not an expected use case for the Python interface.) This should be fixed in a future release.

3.1.3. runSimulation

```
result = xyceObj.runSimulation()
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_runSimulation method of the XyceCInterface. The return value (in result) for this Python method is the same as for the xyce_runSimulation method of XyceCInterface.

3.1.4. simulateUntil

```
(result, actual_time) = xyceObj.simulateUntil(requested_time)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_simulateUntil method of the XyceCInterface. The return value (in result) is the same as for the xyce_simulateUntil method of XyceCInterface. See Section 2.1.4 for a discussion of the requested_time parameter. For the Python method, actual_time is a returned value rather than a parameter in the function call. It is also described in Section 2.1.4.

3.1.5. close

```
xyceObj.close()
```

This method causes Xyce to close all output files after a simulation run is complete and emit timing information. It also deletes the pointer to the N_CIR_Xyce object. It should be called after the Xyce simulation is complete.

3.1.6. getNumDevices

```
(result, numDevices, maxDeviceNameLength) = xyceObj.getNumDevices(modelGroupName)
```

This method assumes that `xyceObj` was previously obtained with the `xyce_interface` method and successfully initialized with the `initialize` method. This method allows the calling Python program to invoke the underlying `xyce_getNumDevices` method. The return value (in `result`) is the same as for the `xyce_getNumDevices` method of `XyceCInterface`. Valid values for the `modelGroupName` parameter are discussed in Section 2.1.7. This method is used internally by the `getDeviceNames`, `getDACDeviceNames` and `getADCMap` methods to pre-allocate the correct-sized arrays for those methods' returned parameter(s) such as `deviceNames`.

3.1.7. getDeviceNames

```
(result, deviceNames) = xyceObj.getDeviceNames(modelGroupName)
```

This method assumes that `xyceObj` was previously obtained with the `xyce_interface` method and successfully initialized with the `initialize` method. This method allows the calling Python program to invoke the underlying `xyce_getDeviceNames` method of the `XyceCInterface`. The return value (in `result`) is the same as for the `xyce_getDeviceNames` method of `XyceCInterface`. For the Python method, `deviceNames` is a returned array rather than a parameter in the function call. Valid values for the `modelGroupName` parameter are discussed in Section 2.1.7.

3.1.8. getDACDeviceNames

```
(result, DACnames) = xyceObj.getDACDeviceNames()
```

This method is basically a specialized version of the Python method `getDeviceNames` that only returns the names of YDAC devices in the simulation. See Section 2.1.4. for more details on the underlying `xyce_getDACDeviceNames` method of `XyceCInterface`.

3.1.9. updateTimeVoltagePairs

```
result = xyceObj.updateTimeVoltagePairs(DACname, timeArray, voltageArray)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_updateTimeVoltagePairs method of the XyceCInterface. The return value (in result) is the same as for the XyceCInterface method xyce_updateTimeVoltagePairs.

The "error condition" of timeArray and voltageArray being of unequal lengths is checked when this method is invoked via the Python interface. If that check fails then this method returns -1. This error condition is not checked when xyce_updateTimeVoltagePairs is invoked directly.

An example of how to use this Python method is provided in the release src subdirectory utils/XyceCInterface/Python_examples/runCircuitWithDACs.

3.1.10. checkResponseVar

```
result = xyceObj.checkResponseVarName(variable_name)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_checkResponseVar method of the XyceCInterface. The return value (in result) is the same as for the XyceCInterface method xyce_checkResponseVar.

3.1.11. obtainResponse

```
(result, value) = xyceObj.obtainResponse(variableName)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_obtainResponse method of the XyceCInterface. The return value (in result) is the same as for the XyceCInterface method xyce_obtainResponse. See Section 2.1.11 for a description of value.

3.1.12. setADCWidths

```
result = xyceObj.setADCWidths(ADCnames, width)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_setADCWidths method of the XyceCInterface. The return value (in result) is the same as for the XyceCInterface method xyce_setADCWidths.

See Section 2.1.12 for a description of the ADCnames and widths parameters. The ADC widths can be set via this function, the WIDTH instance parameter for each individual YADC device and the associated YADC model parameters (see Section 5.1). The order of precedence is in that order. This function should have the highest precedence, since it occurs after the xyce_initialize method is called.

The "error condition" of ADCnames and widths being of unequal lengths is checked when this method is invoked via the Python interface. If that check fails then this method returns -1. This error condition is not checked when xyce_setADCWidths is invoked directly.

3.1.13. getADCWidths

```
(result, width) = xyceObj.getADCWidths(ADCnames)
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_getADCWidths method of the XyceCInterface. See Section 2.1.13 for a description of the ADCnames and widths parameters.

For the Python method, width is a returned array rather than a parameter in the function call. The return value (in result) is the same as for the xyce_getADCWidths method of XyceCInterface. If a given ADC is not found in the netlist then its width value will be returned as 0.

3.1.14. getTimeVoltagePairsADC

```
(result, ADCnames, numADCnames, numPoints, timeArray, voltageArray) =
       xyceObj.getTimeVoltagePairsADC()
```

This method assumes that xyceObj was previously obtained with the xyce_interface method and successfully initialized with the initialize method. This method allows the calling Python program to invoke the underlying xyce_getTimeVoltagePairsADC method of the XyceCInterface. The return value (in result) is the same as for the XyceCInterface method xyce_getTimeVoltagePairsADC.

This function is the "least mature" of the Python methods and Section 7.1 describes its limitations via a Python-based example. Many of these limitations stem from known limitations in the YADC device (see Section 5.1) implemented in Xyce 6.11. In addition, because of an open bug, the use of both getTimeVoltagePairsADC and getTimeStatePairsADC in the same Python program is not recommended.

3.1.15. getTimeStatePairsADC

```
(result, ADCnames, numADCnames, numPoints, timeArray, stateArray) =
       xyceObj.getTimeStatePairsADC()
```

This method assumes that `xyceObj` was previously obtained with the `xyce_interface` method and successfully initialized with the `initialize` method. This method allows the calling Python program to invoke the underlying `xyce_getTimeStatePairsADC` method of the `XyceCInterface`. The return value (in `result`) is the same as for the `XyceCInterface` method `xyce_getTimeStatePairsADC`.

The returned `stateArray` contains the integer-valued states (ranging from 0 to $2 * width[i] - 1$, where $width[i]$ is the bit width of the i^{th} ADC) at the time points specified in the returned `timeArray`. So, this function partially removes the need for an external simulation program to convert the "deltaV" values, stored internally within Xyce, into state values. However, that external simulator may still need to convert the returned integer state-values into a suitable binary/octal/hexadecimal format for its use.

Because of an open bug, the use of both `getTimeVoltagePairsADC` and `getTimeStatePairsADC` in the same Python program is not recommended.

3.1.16. getADCMap

```
(status, ADCnames, widths, resistances, upperVLimits, lowerVLimits, settlingTimes) =
 xyceObj.getADCMap()
```

This method assumes that `xyceObj` was previously obtained with the `xyce_interface` method and successfully initialized with the `initialize` method. This method allows the calling Python program to invoke the underlying `xyce_getADCMap` method of the `XyceCInterface`. See Section 2.1.16 for a description of the `ADCnames`,`widths`, `resistances`, `upperVLimits`, `lowerVLimits` and `settlingTimes` returned arrays.

The return value (in `result`) is the same as for the `XyceCInterface` method `xyce_getADCMap`. If the netlist has no ADC devices then returned arrays will be empty.

3.2. EXAMPLES

This section gives a brief example of how to run a Xyce simulation from a Python (2.6 or 2.7) program using the Sandia-supplied ctypes-based interface. Since Python is an interpreted langauge there is no need for further compilation or linking of Xyce. It is sufficient to have built Xyce as "shared objects" per the instructions in Section 2.2.

An example Python program, called `runACircuit.py`, is shown in Figure 3-1. The associated Xyce netlist, which is called runACircuit.cir, is shown in Figure 3-2. (Note: These files are also found in the release src subdirectory `utils/XyceCInterface/Python_examples` just in case cut-n-paste from the .pdf document does not work for the reader.) That Python program can then be invoked with:

```
python runACircuit.py </path/to/where/libxycecinterface/is/installed>
```

The one caveat is that the location of `xyce_interface.py` should be added to your `PYTHONPATH` environment variable. The file `UpdatePythonPath.sh` in the release subdirectory `utils/XyceCInterface` provides a "non-working" example of how to modify that environment variable. The path (`/path/to/XyceSrcDirectory/utils/XyceCInterface`) in that file should be replaced with the actual path to your Xyce source directory.

For the Xyce 6.10 release, we recommended the use of the `LD_LIBRARY_PATH` environment variable as a means of communicating the location of the .so files to the Python interface code. That approach meant that the Python interface would not work on newer versions of OSX because of Apple's System Integrity Protection (SIP) feature. To work around that issue, the invocation line must now include the location where the shared object files are installed. If Xyce was built and installed per the instructions in Section 2.2 then that location will be `$installDir/lib`

Additional examples of using `xyce_interface.py` can be found in the release src subdirectory `utils/XyceCInterface/Python_examples`. Those examples also use the `simulateUntil()`, `getDACDeviceNames()`, `updateTimeVoltagePairs()` and `obtainResponse()` methods. See Reference [10] for an example of how to use `xyce_interface.py` to interface Xyce to GHDL [11] and Cocotb [12].

Additional examples can also be found in the Xyce regression test suite in the subdirectory `Netlists/MIXED_SIGNAL/Python`. However, some of those examples are "error condition" tests, which purposefully fail or otherwise have purposefully invalid or non-useful syntaxes. The comments in the files for each test should indicate which ones are functional examples and which lines in a given test are not valid or useful.

For internal High Performance Computing (HPC) users, the .so files needed to run these examples can be found in `/projects/xyce/XyceRad_6.11/Serial/toss3/lib`. The Python interface file is then in `/projects/xyce/XyceRad_6.11/Serial/toss3/python`. The examples are in `/projects/xyce/XyceRad_6.11/Serial/toss3/examples/Python_examples`.

For internal Common Engineering Environment (CEE) users, the .so files needed to run these examples can be found in (where `RHELX` is either `RHEL6` or `RHEL7`, depending on the CEE machine you are using) `/projects/xyce/Xyce_6.11/RHELX/Serial/lib`. The Python interface file is then in `/projects/xyce/Xyce_6.11/RHELX/Serial/python`, and the `UpdatePythonPath` script has the correct path for each RHEL version. The examples are in `/projects/xyce/Xyce_6.11/RHELX/Serial/examples/Python_examples`. The "path to where libxycecinterface is installed" is `/projects/xyce/Xyce_6.11/RHELX/Serial/lib`.

```
import sys
from xyce_interface import xyce_interface

# this calls the xyce_interface.open() method to
# make a xyce object
libDirectory = sys.argv[1]
xyceObj = xyce_interface(libdir=libDirectory)
print( xyceObj )

argv= ['runACircuit.cir']
print( "calling initialize with netlist %s" % argv[0] )

result = xyceObj.initialize(argv)
print( "return value from initialize is %d" % result )

print( "Calling runSimulation..." )
result = xyceObj.runSimulation()
print( "return value from runSimulation is %d" % result )

print( "calling close")
xyceObj.close()
```

Figure 3-1. Python Program for runACircuit example

```
* test circuit
V1 1 0 SIN(0 1 1)
R1 1 0 1

.TRAN 0 1
.PRINT TRAN V(1)
.MEASURE TRAN MAXV1 MAX V(1)
.MEASURE TRAN MINV1 MIN V(1)

.END
```

Figure 3-2. Xyce Netlist for runACircuit Python example

3.3. KNOWN LIMITATIONS AND BUGS

This section has a list of the known limitations and bugs of the Python-based version of the Mixed Signal interface.

- This interface has been tested with Python 2.6.6, 2.7.4 and 2.7.5 for the Xyce 6.11 release. Some features are known to fail when this interface is used with Python 3.4.2 or 3.5.2. Support for Python 3 will likely be added in a future release.
- The `getTimeVoltagePairsADC()` and `getTimeStatePairsADC()` methods are currently limited to returning only up to 1000 devices each. The individual device names must each also be less than 1000 characters long. There is also a limit of 1000 time/state points that can be returned by those methods. The memory management for these two methods will likely be improved in the next release, in conjunction with fixes for the "Coordinated Time Stepping Issues" mentioned in Section 7.1. In the meantime, the work-around is to manually edit the source for those methods in `xyce_interface.py`.
- The use of both `getTimeVoltagePairsADC` and `getTimeStatePairsADC` in the same Python program is not recommended, because of an open bug.
- If the `initialize()` method returns 2 (which is the Xyce::Circuit::Simulator::RunStatus of "DONE") then the calling .py file will likely segfault. This can happen for Xyce command line options such as `-norun` that prevent Xyce from proceeding to a full simulation. This is not an expected use case for the Python interface though.

4. XYCE VPI INTERFACE TO ICARUS

This chapter describes how the `XyceCInteface` class can be used to interface Xyce to Icarus, which is an open-source Verilog simulation and synthesis tool. It begins with a brief overview of Icarus. It then gives a working "runXyce" example where a Xyce simulation is called from a simple Verilog program via the `vvp` executable produced by Icarus. It concludes with guidance on building the example `runXyce.vvp` and `runXyce.vpi` files.

4.1. ICARUS OVERVIEW

Since this application note assumes minimal familiarity with Verilog and Icarus, some helpful references for Icarus are:

- Icarus Verilog Home Page [2]
- Download and Build Instructions [13]
- Getting Started [14]
- VPI Example [15]

The next two subsections assume that the reader has downloaded and installed Icarus according to those Download and Build Instructions. It also assumes that the reader can execute the simple "Hello World" examples given at those Getting Started and VPI Example webpages.

For more information on VPI, consult the IEEE Standard [3]. This book [16] also has a good set of VPI examples, with example code.

4.2. XYCE VPI IMPLEMENTATION AND EXAMPLES

As mentioned previously, this is the initial implementation of a Verilog Procedural Interface (VPI) capability for Xyce. It is subject to change in future Xyce releases. In particular, this initial version accesses the `XyceCInteface` class directly within the VPI code. Subsequent versions will likely use a "C++ wrappers" approach so that the VPI code only uses ANSI-C and the native PLI data-types in its function calls.

This section describes how to use the `XyceCInterface` class to run a Xyce simulation from a Verilog program via the VPI capability supported by Icarus. This is a very simple demonstration of that interface that is basically a "runXyce" example that uses a Verilog program (`runXyce.v`),

a Xyce netlist (`runXyce.cir`) and some VPI code (`runXyce.c`), as shown in Figures 4-1, 4-2 and 4-3. It is basically the same as the runACircuit example given in Section 3.2. (Note: all three of these files can be also found in the release src subdirectory `utils/XyceCInterface/VPI_examples/runXyce`.)

Additional examples of using the VPI interface with Icarus can be found in the release src subdirectory `utils/XyceCInterface/VPI_examples`. Those examples also use the `xyce_simulateUntil()`, `xyce_getDACDeviceNames()`, `xyce_updateTimeVoltagePairs()` and `xyce_obtainResponse()` methods of the `XyceCInterface`.

```
module main;
initial $runXyce;
endmodule
```

Figure 4-1. Verilog Program for runXyce VPI example

```
* test circuit
V1 1 0 SIN(0 1 1)
R1 1 0 1

.TRAN 0 1
.PRINT TRAN V(1)
.MEASURE TRAN MAXV1 MAX V(1)
.MEASURE TRAN MINV1 MIN V(1)

.END
```

Figure 4-2. Xyce Netlist for runXyce VPI example

```
#include <vpi_user.h>
#include <stdio.h>
#include <stdlib.h>
#include <N_CIR_XyceCInterface.h>
static int runXyce_compiletf(char*user_data) {
  return 0;
}
static int runXyce_calltf(char*user_data) {
  // Used as a pointer to a pointer to an N_CIR_Xyce object.
  // This somewhat convoluted syntax is needed to stop p from
  // pointing at the same address as the VPI system task.
  void** p = (void **) malloc( sizeof(void* [1]) );

  // Make Xyce command line for xyce_initialize() call.
  char *argList[] = {(char*)("Xyce"),(char*)("runXyce.cir") };
  int argc = sizeof(argList)/sizeof(argList[0]);
  char** argv = argList;

  // Demo methods in utils/XyceCInterface/N_CIR_XyceCInterface.C
  xyce_open(p);
  xyce_initialize(p,argc,argv);
  xyce_runSimulation(p);
  xyce_close(p);

  // pointer clean-up and return
  free(p);
  return 0;
}
void runXyce_register() {
  s_vpi_systf_data tf_data;
  tf_data.type      = vpiSysTask;
  tf_data.tfname    = "$runXyce";
  tf_data.calltf    = runXyce_calltf;
  tf_data.compiletf = runXyce_compiletf;
  tf_data.sizetf    = 0;
  tf_data.user_data = 0;
  vpi_register_systf(&tf_data);
}
void (*vlog_startup_routines[])() = {
  runXyce_register,
  0 /* final entry must be zero */
};
```

Figure 4-3. VPI File for runXyce VPI example

4.3. VPI BUILDING GUIDE FOR RHEL6 AND RHEL7

The sequence of commands shown in Figure 4-4 should compile Icarus and the Xyce shared objects into an executable `vvp` program. (Note: This process was tested with Icarus Verilog version 11.0.) It is analogous to the compilation steps given on the Icarus VPI Example web page [15]. In this command sequence the top-level Xyce build directory build is denoted as `$xyceBuildDir` and the top-level Xyce src directory is referred to as `$xyceSrcDir`. `$verilogBase` can be generated by running `which iverilog` and using the returned directory path starting above the `bin` subdirectory. `$baseName` is then the common prefix (e.g., `runXyce`) of the .c and .v. files.

The sequence of commands shown in Figure 4-4 assumes that Xyce built according to the instructions in Section 2.2. After the `runXyce.vvp` program is made then it can be executed with:

```
vvp -M. -mrunXyce runXyce.vvp
```

```
gcc -c -fpic -I$verilogBase/include/iverilog \
-I$xyceSrcDir/utils/XyceCInterface \
-I$xyceBuildDir/include \
$baseName.c

gcc -I$verilogBase/include/iverilog/libvpi.a \
-shared -L$xyceBuildDir/lib \
-L$xyceBuildDir/utils/XyceCInterface/.libs -lxycecinterface \
-Wl,-rpath=$xyceBuildDir/lib \
-Wl,-rpath=$xyceBuildDir/utils/XyceCInterface/.libs \
-o $baseName.vpi $baseName.o

iverilog -o$baseName.vvp $baseName.v
```

Figure 4-4. Compiling vvp Program with an Installed Xyce Build

Finally, additional examples of using Xyce with Icarus and VPI can be found in the release subdirectory `utils/XyceCInterface/VPI_examples`. Those examples also use the `xyce_simulateUntil()`, `xyce_getDACDeviceNames()`, `xyce_updateTimeVoltagePairs()` and `xyce_obtainResponse()` methods of the `XyceCInterface` class.

5. DEVICE MODELS FOR MIXED SIGNAL SIMULATION

Xyce has simple models for a Digital-to-Analog Converter (DAC) and an Analog-to-Digital Converter (ADC) that help demonstrate the Python and VPI interfaces discussed in the previous chapters. These models will likely be enhanced in future releases, so feedback on missing features is encouraged.

This chapter contains manual pages for the YADC and YDAC devices. This information may be moved to the Xyce Reference Guide in a future release.

5.1. ANALOG-TO-DIGITAL CONVERTER

Instance Form `YADC<name> <(+) node> <(-) node> [model name] [device parameters]`

Model Form `.MODEL <model name> ADC [model parameters]`

Examples

```
YADC1 ADC1 1 2 simpleADC R=1T WIDTH=2
.MODEL simpleADC ADC (settlingtime=50ns uppervoltagelimit=5
+ lowervoltagelimit=0)
```

Parameters and Options

`(+) node`
`(-) node`
: Polarity definition for a positive voltage across the ADC. The first node is defined as positive. Therefore, the voltage across the component is the first node voltage minus the second node voltage.

`model name`
: This parameter is optional for the YADC device. If it is omitted then the default values for the model parameters will be used.

`device parameters`
: Parameters listed in Table 5-1 may be provided as space separated `<parameter>=<value>` specifications as needed. Any number of parameters may be specified.

Comments

The "upper voltage limit"and "lower voltage limit" model parameters might not be the best approach for this device. They might be replaced, in a future release, with a `Vref+` node against which a Vin is compared, with a common negative reference (e.g. ground). For now, a reasonable approach is to connect the negative terminal to ground and use 0.0 as the value for the `LOWERVOLTAGELIMIT` parameter.

The YADC device is calculating "breakpoints" for when its output digital states change. However, there are at least two known issues with that process in this Xyce release. First the breakpoint times (and the voltage difference between the positive and negative terminals at those times) are based on the times of "accepted steps" in the simulation, rather than (possibly) interpolated estimates of when the state changes actually occurred. So, those times and voltages may be inaccurate, and be reported as occuring later than the actual state-change times. The second, and larger issue, is that breakpoints calculated by the YADC device are not actually used by the rest of the Xyce simulation. Instead, those times and voltages are simply made available to the external simulator via the `xyce_getTimeVoltagePairsADC` method described in Section 2.1.14.

The YADC device stores "deltaV" (the difference between the voltages at the positive and negative terminals) in the TVVEC time-voltage vector returned by the `XyceCInterface` method `xyce_getTimeVoltagePairsADC`. So, the external digital simulator has two options. It can duplicate the calculation of the output state of the YADC device, via the equations given below. Or it can request the integer-value state directly via the `xyce_getTimeStatePairsADC` method. However, because of an open bug, only one of those methods should be used in a given program.

The capability to output the YADC device-state directly via the `N()` operator on a `.PRINT` line has been added for Xyce 6.11. A syntax example is as follows where `YADC!ADC1` is the fully-qualified name in this example:

```
YADC ADC1 1 0 simpleADC R=1T
.model simpleADC ADC(settlingtime=50ns uppervoltagelimit=2
+ lowervoltagelimit=0 width=3)
.PRINT TRAN N(YADC!ADC1_STATE)
```

A final issue may be that the output state is defined as an integer, between 0 and 2**WIDTH-1. An option to have it reported, to the external simulators, as a binary bit-vector might be added in a future release.

Table 5-1. ADC Device Instance Parameters

Parameter	Description	Units	Default
R	Internal resistance	Ω	1e+12
WIDTH	Output bit vector width	s	1

Table 5-2. ADC Device Model Parameters

Parameter	Description	Units	Default
LOWERVOLTAGELIMIT	Lower limit of ADC voltage range	V	0
SETTLINGTIME	Settling time	s	1e-08
UPPERVOLTAGELIMIT	Upper limit of ADC voltage range	V	5
WIDTH	Output bit vector width	s	1

ADC Equations C++ style code for how the output state of the YADC device is calculated is shown in Figure 5-1. (Note: this code comes from `Instance::getInstanceBreakPoints()` in the source file src/DeviceModelPKG/OpenModels/N_DEV_ADC.C.)

```
// vPos is the voltage on the positive terminal.
// vNeg is the voltage on the negative terminal.
// width_ is the Output bit vector width (from WIDTH).
// nQuantLevels_ is 2**(width_).
deltaV = vPos-vNeg;
vFrac = deltaV/(model_.upperVoltageLimit_
                  - model_.lowerVoltageLimit_);

if (vFrac < (1.0)/(nQuantLevels_) )
{
  newState = 0;
}
else if (vFrac >= (nQuantLevels_ - 1.0)/(nQuantLevels_))
{
  newState = nQuantLevels_ -1;
}
else
{
  newState =  int(vFrac*nQuantLevels_);
}
if (newState != lastOutputLevel_)
{
  // update TVVEC with deltaV value and breakpoint time
}
```

Figure 5-1. Calculation of the YADC Output State

5.2. DIGITAL-TO-ANALOG CONVERTER

Instance Form `YDAC<name> <(+) node> <(-) node> [model name]`

Model Form `.MODEL <model name> DAC [model parameters]`

Examples

```
YDAC dac1 2 0 simpleDAC
.model simpleDAC DAC (tr=5e-9 tf=5e-9)
```

Parameters and Options

`(+) node`
`(-) node`
Polarity definition for a positive voltage across the DAC. The first node is defined as positive. Therefore, the voltage across the component is the first node voltage minus the second node voltage.

`model name`
This parameter is optional for the DAC device. If it is omitted then the default values for the model parameters will be used.

Comments The DAC device acts like a voltage source as far as the rest of the circuit is concerned. There is no output R-L-C smoothing network, as might be found in a more realistic DAC.

Table 5-3. DAC Device Model Parameters

Parameter	Description	Units	Default
TF	Fall Time	s	1e-09
TR	Rise Time	s	1e-09

6. SUMMARY OF CHANGES SINCE XYCE 6.10

This chapter provides a brief summary of the changes in the Mixed Signal Interface since Release 6.10 of Xyce.

- The syntax for invoking the Python interface has changed. See Section 3.2 for more details.
- The XyceCInterface methods `xyce_simulateUntil`, `xyce_getADCMap`, `xyce_obtainResponse`, `xyce_getDeviceNames` and `xyce_getDACDeviceNames` have been modified to improve their C-langauge compliance. They now use double* and int*, rather than double& or int&. The latter is allowed in C++, but not in C.
- The recommended steps for compiling an executable vvp program, for the VPI interface, has changed. See Section 4.3 for more details.
- Two new methods, `xyce_getADCWidths` and `getADCWidths`, were added to the XyceCInterface and the Python wrappers to that interface. They allow the user to get the current values of the `WIDTH` instance parameter at a specified list of ADCs in the netlist.
- Two new methods, `xyce_getADCMap` and `getADCMap`, were added to the XyceCInterface and the Python wrappers to that interface. They allow the user to get the current values of all of the instance and model parameters at every ADC in the netlist.
- Two new methods, `xyce_getTimeStatePairsADC` and `getTimeStatePairsADC` were added to the XyceCInterface and the Python wrappers to that interface. They return the ADC state as an integer, rather than the "deltaV" value.
- The `WIDTH` parameter is now both an instance parameter and a model parameter for the ADC device.
- The XyceCInterface methods `xyce_getDeviceNames` and `xyce_getDACDeviceNames`, and their corresponding Python wrappers, have improved return codes that now properly indicate success or failure for those calls. Success in this context means that at least one device of the requested type exists in the netlist.
- The "error condition" of the `timeArray` and `voltageArray` being of unequal lengths is now checked when the Python method `updateTimeVoltagePairs` is used.
- The "error condition" of `ADCnames` and `widths` being of unequal lengths is now checked when the Python methods `getADCWidths` and `setADCWidths` are used.

- The method `getDeviceNames()` might not work correctly if an invalid model group name was used as a parameter. In that case, the first letter of the model group name would be used. So, for example, a request for devices of model group `BOGO` would actually return all of the B devices in the netlist. This is fixed now, and a request for model group `BOGO` will now return an error.
- The XyceCInterface method `xyce_updateTimeVoltagePairs` used to unconditionally emit the warning message "Netlist warning: Failed to update the timevoltage pairs for the DAC" even if the update was successfully. This has been fixed.

7. CONCLUSIONS AND FUTURE WORK

This application note provided an overview of the `XyceCInterface` class and how it can used to interface to external programs via a Sandia-supplied Python ctypes interface and the Verilog Procedural Interface (VPI). These interfaces are not an "officially announced" capability in Xyce yet. So, one purpose of this application note was to solicit feedback on these interfaces from both internal Sandia Xyce users and other performers on the DARPA Posh Open Source Hardware (POSH) program. The remainder of this chapter will summarize the known limitations of these interfaces.

The Common Operating Environment (COE) at Sandia encourages internally developed software to support RHEL6, RHEL7, OSX and Windows 10. In addition, support for Ubuntu may be part of that COE in the near future. At present, the interfaces described in this application note have only been tested and documented for RHEL6 and RHEL7.

A list of miscellaneous bugs for the Python interface was listed in Section 3.3. The main issue with that Python interface, which is "coordinated time stepping", will be discussed in the next subsection.

The primary issue with the VPI capability is the lack of standards compliance. The example given in Section 4.2 uses the C++ features of the `XyceCInterface` directly. Wrapper functions, that only use ANSI C and the native PLI data-types in their function calls, still need to be implemented.

7.1. KNOWN ISSUES WITH COORDINATED TIME STEPPING

The `xyce_getTimeVoltagePairsADCmethod` of the `XyceCInterface` is the least mature of that interface's methods. Many of its limitations stem from known limitations in the YADC device (see Section 5.1) implemented in Xyce 6.11. This section gives a Python-based example that illustrates those limitations. The goal is to solicit feedback on the best resolution of these issues.

The netlist for this "TimeStepping" example is shown in Figure 7-1. The calling Python program is shown in Figure 7-2. An abbreviated version of the resultant stdout, with a subset of the descriptive output from the Python program is then shown in Figure 7-3.

The returned arrays (`timeArray` and `voltageArray`) are 2x2 in this example. In general, they would be MxN where is the value of `numADCnames` and N is the value of `numPoints`. For the simulation interval ending at 1e-5, the returned values of (0,0) are "not useful". They are basically

the simulation start time. The returned values of (1e-5,2e-1) are also not useful in this case. They are the breakpoints set by the call to `simulateUntil`. So, the underlying `xyce_getTimeVoltagePairsADC()` method of the `XyceCInterface`, and the device model in the YADC device, may need to be modified to only report breakpoints that were set by the ADC devices.

Another problem is accuracy. There is useful breakpoint information returned for `ADC2` after the second call to `simulateUntil`. However, the `time` (1.267e-05) and `value` (2.524e-01) were determined based on the last accepted Xyce time step (see the .prn file) at time = 1.262e-05, instead of when the state change might have actually occured. A related issue is that the returned value is the voltage difference between the positive and negative terminals of the YADC device. So, the external simulator has to duplicate the YADC equations (see Section 5.1) to determine the binary-state value for each YADC in the simulation.

The final, and most important problem, is that the breakpoints generated by each YADC device are not actually used by the rest of the Xyce simulation. The Xyce simulation in this example continued on until the next value of `requested_time` and did not pause at any of the breakpoints generated by the YADC devices. (Note: that capability was broken in a previous release and was not fixed/changed in time for the Xyce 6.11 release.) So, based on various Sandia and DARPA POSH use cases, techniques for coordinated time-stepping of Xyce and the external simulator(s) need to be defined and implemented for simulations that contain both YADC and YDAC devices.

```
* Netlist name is TimeStepping.cir
* These WIDTH values will be overwritten by the Python program
YADC adc1 1 0 simpleADC R=1T WIDTH=1
YADC adc2 1 0 simpleADC R=1T WIDTH=1
.model simpleADC ADC(settlingtime=50ns uppervoltagelimit=2
+ lowervoltagelimit=0)

v1 1 0 PWL 0 0 1e-4 2
.TRAN 0 1e-4

* illustate syntax for printing out YADC device parameters
.PRINT TRAN V(1) YADC!ADC1:WIDTH YADC!ADC2:WIDTH
.END
```

Figure 7-1. Xyce Netlist for Time Stepping Example

```
import sys
from xyce_interface import xyce_interface
libDirectory = sys.argv[1]
xyceObj = xyce_interface(libdir=libDirectory)
print( xyceObj )
argv= ['TimeStepping.cir']
print( "calling initialize with netlist %s" % argv[0] )
result = xyceObj.initialize(argv)
print( "return value from initialize is %d" % result )
# get ADC names
(result, names) = xyceObj.getDeviceNames("YADC")
print( "return value from getDeviceNames is %d" % result )
print( names )
#set ADC widths.  This is hard-coded for two ADCs, and must
# match the WIDTH variables on the ADC instance lines. This may
# seem backwards but names is ['YADC!ADC2', 'YADC!ADC1'] here.
width=[3,2]
result = xyceObj.setADCWidths(names,width)

stepSize = 1e-5
steps = range(0,3)
for i in steps:
  requested_time = 0.0 + (i+1) * stepSize
  print( "Calling simulateUntil for requested_time = %.3e" \
  % requested_time )
  actual_time = 0.0
  (result, actual_time)=xyceObj.simulateUntil(requested_time)
  print( "simulateUntil status = %d and actual_time = %.3e" \
  % (result, actual_time) )
  (result, ADCnames, numADCnames, numPoints, timeArray, \
  voltageArray) = xyceObj.getTimeVoltagePairsADC()
  print( "number of pts returned by getTimeVoltagePairsADC() \
  is %d" % numPoints )
  # Note: ADCnames is ['YADC!ADC1', 'YADC!ADC2'] here.
  print("ADC 1: Time and voltage array 0 values are %.3e %.3e" \
  %(timeArray[0][0] , voltageArray[0][0]) )
  print("ADC 1: Time and voltage array 1 values are %.3e %.3e" \
  %(timeArray[0][1] , voltageArray[0][1]) )
  print("ADC 2: Time and voltage array 0 values are %.3e %.3e" \
  %(timeArray[1][0] , voltageArray[1][0]) )
  print("ADC 2: Time and voltage array 1 values are %.3e %.3e" \
  %(timeArray[1][1] , voltageArray[1][1]) )
print( "calling close")
xyceObj.close()
```

Figure 7-2. Python Program for Time Stepping Example

```
Calling simulateUntil for requested_time = 1.000e-05
simulateUntil status = 1 and actual_time = 1.000e-05
number of pts returned by getTimeVoltagePairsADC()   is 2
names are YADC!ADC1 YADC!ADC2
ADC 1: Time and voltage array 0 values are 0.000e+00 0.000e+00
ADC 1: Time and voltage array 1 values are 1.000e-05 2.000e-01
ADC 2: Time and voltage array 0 values are 0.000e+00 0.000e+00
ADC 2: Time and voltage array 1 values are 1.000e-05 2.000e-01
Calling simulateUntil for requested_time = 2.000e-05
simulateUntil status = 1 and actual_time = 2.000e-05
number of pts returned by getTimeVoltagePairsADC()   is 2
names are YADC!ADC1 YADC!ADC2
ADC 1: Time and voltage array 0 values are 2.000e-05 4.000e-01
ADC 1: Time and voltage array 1 values are 0.000e+00 0.000e+00
ADC 2: Time and voltage array 0 values are 1.267e-05 2.524e-01
ADC 2: Time and voltage array 1 values are 2.000e-05 4.000e-01
Calling simulateUntil for requested_time = 3.000e-05
simulateUntil status = 1 and actual_time = 3.000e-05
number of pts returned by getTimeVoltagePairsADC()   is 2
names are YADC!ADC1 YADC!ADC2
ADC 1: Time and voltage array 0 values are 2.625e-05 5.241e-01
ADC 1: Time and voltage array 1 values are 3.000e-05 6.000e-01
ADC 2: Time and voltage array 0 values are 2.625e-05 5.241e-01
ADC 2: Time and voltage array 1 values are 3.000e-05 6.000e-01
calling close
```

Figure 7-3. Abbreviated stdout for Time Stepping Example

REFERENCES

[1] Thomas V. Russo and Russell Hooper. Application Note: Coupled Simulation with the Xyce General External Interface. Technical Report SAND2018-12275, Sandia National Laboratories, 2018.

[2] *Icarus Home Page*, . URL http://iverilog.icarus.com/.

[3] *IEEE Standard 1364-2005: IEEE Standard for Verilog Hardware Description Language*, 2005.

[4] Eric R. Keiter, Karthik V. Aadithya, Ting Mei, Thomas V. Russo, Richard L. Schiek, Peter E. Sholander, Heidi K. Thornquist, and Jason C. Verley. Xyce Parallel Electronic Simulator: Reference Guide, Version 6.11. Technical Report SAND2019-5950, Sandia National Laboratories, Albuquerque, NM, 2019.

[5] Eric R. Keiter, Karthik V. Aadithya, Ting Mei, Thomas V. Russo, Richard L. Schiek, Peter E. Sholander, Heidi K. Thornquist, and Jason C. Verley. Xyce Parallel Electronic Simulator: Users' Guide, Version 6.11. Technical Report SAND2019-5949, Sandia National Laboratories, Albuquerque, NM, 2019.

[6] Eric R. Keiter, Scott A. Hutchinson, Robert J. Hoekstra, Lon J. Waters, and Thomas V. Russo. Xyce parallel electronic simulator design: Mathematical formulation, version 2.0. Technical Report SAND2004-2283, Sandia National Laboratories, Albuquerque, NM, June 2004.

[7] *Xyce Building Guide*, . URL https://xyce.sandia.gov/documentation/BuildingGuide.html.

[8] *Running the Xyce Regression Suite*, . URL https://xyce.sandia.gov/documentation/RunningTheTests.html.

[9] *ctypes - A foreign function library for Python*. URL https://docs.python.org/3/library/ctypes.html.

[10] Andrew M. Smith, Jackson Mayo, Rob Armstrong, Richard Schiek, Peter Sholander, and Ting Mei. Digital/Analog Cosimulation Using CocoTB and Xyce. Technical Report SAND2018-TBD, Sandia National Laboratories, 2018.

[11] *GHDL*. URL http://ghdl.free.fr/.

[12] *Introduction to Cocotb*. URL https://cocotb.readthedocs.io/en/latest/introduction.html.

[13] *Icarus Verilog - Installation Guide*, . URL http://iverilog.wikia.com/wiki/Installation_Guide.

[14] *Icarus Verilog - Getting Started*, . URL http://iverilog.wikia.com/wiki/Getting_Started.

[15] *Icarus Verilog - Using VPI*, . URL http://iverilog.wikia.com/wiki/Using_VPI.

[16] Stuart Sutherland. *The Verliog PLI Handbook: Second Edition*. Springer Science+Business Media, New York, NY, 2002.

DISTRIBUTION

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.