HyperChem® for Windows and NT

The Chemist's Developer Kit (CDK):

Customizing HyperChem

Interfacing to HyperChem

Hypercube, Inc.

Publication HC50-00-04-00 October 1996

PRINTED IN CANADA

Table of Contents

Chapter 1
Introduction

The Chemist's Developer Kit

The Chemist's Developer Kit (CDK) allows you to:

- Customize HyperChem For Your Own Purposes
- Interface Your Own Programs to HyperChem

CDK for Windows or NT

This manual describes the Chemist's Developer Kit (CDK) to be used in association with Release 5.0 or greater of HyperChem® for Windows and NT. The CDK allows you to *customize* HyperChem for your own special purposes, by modifying its menus and introducing scripts that each custom menu item can invoke. Alternatively, you can use the CDK to *interface* HyperChem to external programs written in Visual Basic, C, Fortran, etc. The CDK for Windows and NT, as described here, allows you to extend HyperChem in a multitude of ways but always within the confines of Microsoft Windows 95 (or greater) or Windows NT 3.51 (or greater).

Equivalent Unix CDK

A similar but somewhat modified CDK and manual will be made available for Unix. That manual will describe how to, correspondingly, customize Unix versions of HyperChem and how to interface a UNIX version of HyperChem to programs written for Unix. It will also describes how to interface a Windows or NT version of HyperChem to your Unix program.

Components of the CDK

The CDK is included as a part of the general Release 5.0 of HyperChem for Windows 95 and NT. Example scripts and programs are found in sub-directories on the HyperChem CD-ROM associated with the the relevant programming environment - C, Fortran, etc. The current CDK has only limited capability with older versions of HyperChem.

Components Included with Release 5

The CDK consists essentially consists of:

- A HyperChem Executable (Release 5.0 or greater)
- This Manual
- Example Custom Menu Files (`*.MNU`)
- Example Hcl Scripts (`*.SCR`)
- Example Tcl/Tk Scripts (`*.TCL`)
- The HyperChem Application Programming Interface Library (`HC.H`, `HCLOAD.C`, `HSV.H`, `HAPI.DLL` and `HAPI.LIB for C/C++ environments`)
- Related HAPI Files for Other Languages
- Example Programs Interfaced to HyperChem

Other Suggested Tools

To customize HyperChem no other tools are required. However, to explore interfacing HyperChem to other Windows programs, one or more of the following may be useful:

- Microsoft Word
- Microsoft Excel
- Microsoft Visual Basic

These three have been used in this manual to illustrate the low-level DDE interfaces to HyperChem described in Chapters 8 and 9. Other Windows word processors (for example, WordPerfect) or spreadsheets (for example, Quattro Pro) could be used in place of Word and Excel but the authors have little experience with them. Visual Basic is preferred by the authors as a rapid

prototyping language for preparing a visual interface to HyperChem. Again, other such visual tools are available from Borland and other manufacturers.

Visual Basic may also be used with the higher-level HyperChem Application Programming Interface (HAPI) replacing the lower-level DDE interface.

Suggested Compilers

To explore compiled C, C++, or Fortran programs interfaced to HyperChem, appropriate compilers and development environments are required. We have used

- Microsoft Visual C++ 4.0
- Microsoft Fortran PowerStation 4.0

Equivalent compiler tools from other manufacturers, such as Watcom, may be used but the examples of the CDK have only been tested with the Microsoft tools.

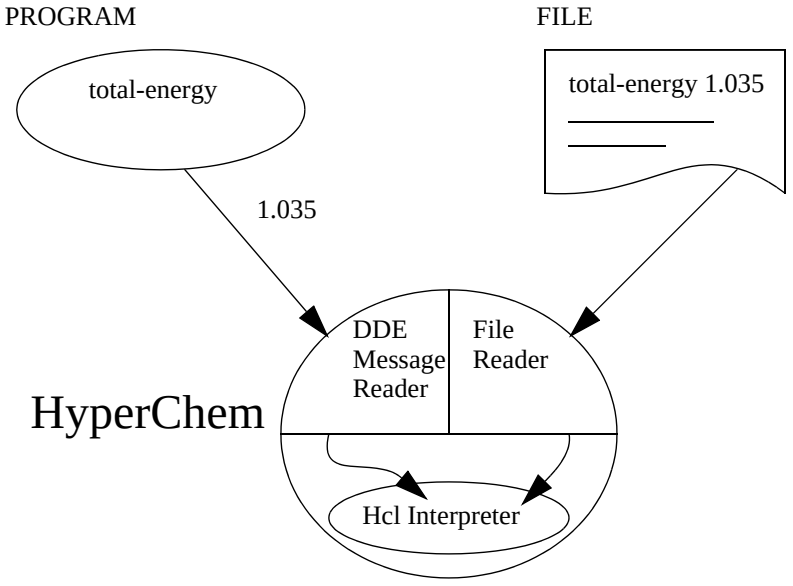
HyperChem State Variables

The CDK and much of what follows are possible because HyperChem implements the concept of a *state variable*. A HyperChem State Variable (HSV) is one of hundreds of variables and data structures that is registered at the instantiation of HyperChem and is henceforth available for flexible and robust reading and writing at any later time. An example of an HSV is the total energy of the system which is represented by the hyphenated string,

```
total-energy
```

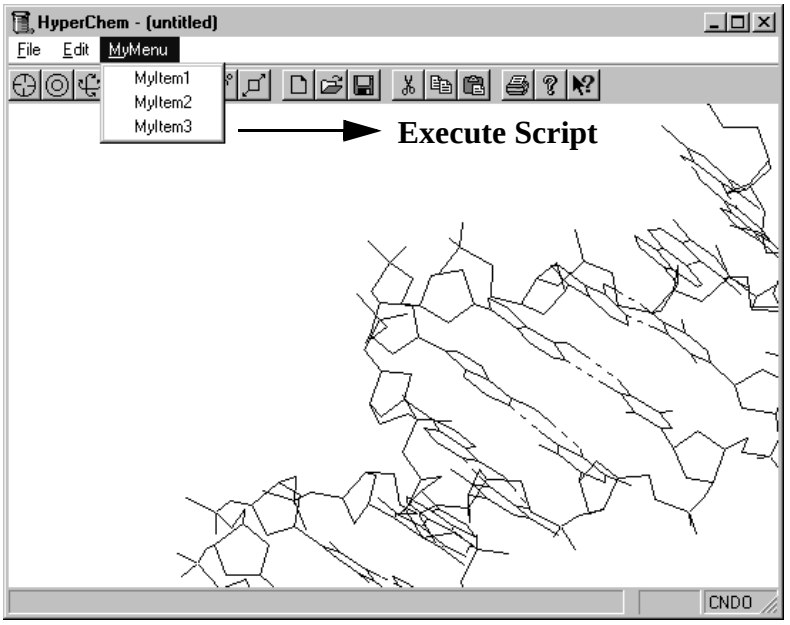
At any time, even when the system still has no molecule or no calculation has yet been performed on the molecular system (and the total energy as the ini-

tialized value zero), the HSV, *total-energy*, is available for reading and writing, either within a HyperChem script or from an external program.



Customizing HyperChem

HyperChem has two quite unique features that allow you to customize it. The first of these is that it has its own set of *script commands* that can activate essentially any of the program's functionality. The second is that HyperChem allows its whole menu structure to be replaced by *custom menus*.



Internal Script commands

Script commands each consist of a single line of text, such as *do-molecular-dynamics* and *menu-file-open*, that invoke HyperChem actions or read and write HyperChem State Variables (HSV's) via a Scripting Interface (SI) that is a superset of the interactive Graphical User Interface (GUI) available via mouse and keyboard.Thus a molecular dynamics trajectory can be initiated by mouse clicks on the appropriate menu items and dialog boxes or by the above script commands. Alternatively, one can bring up the <File/Open...>[1]dialog box by clicking on the appropriate menu item or by executing the appropriate script command, *menu-file-open*. Script commands can be of two types, Hcl or Tcl/Tk.

1. Here and throughout the text we will periodically use angular brackets to delimit menu items (and other text) so as not to confuse it with surrounding text.

HyperChem Command Language (Hcl)

Scripts can be written in the *HyperChem Command Language (Hcl)*, pronounced “hickle”, that has been part of HyperChem since its first public release. These scripts, now referred to in Release 5.0 as Hcl scripts consist of a simple sequence of Hcl commands (strings), such as the *do-molecular-dynamics* script command referred to above. Hcl scripts are stored in `*.SCR` files that contain only straight line code consisting of a sequence of single line Hcl commands. Hcl scripts contain no Variables or Control Statements such as IF, ELSE, DO, etc.

Tool Command Language (Tcl/Tk)

With Release 5.0, HyperChem can now execute a *Tcl/Tk Script* that consist of Hcl script commands embedded inside a conventional Tcl/Tk script. A Tcl/Tk script is one that is interpreted by the well known public domain Tcl command interpreter that is now part of HyperChem Release 5.0. The Tcl interpreter allows a rich control structure of variables, loops, conditionals, etc. It is augmented by a tool kit (Tk) that allows a Tcl script to define new graphical elements, such as dialog boxes, allowing you to extend HyperChem’s GUI.

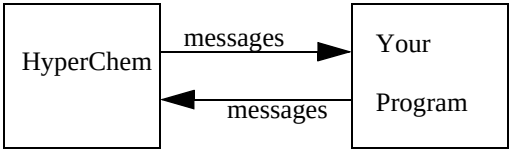
Custom Menus

HyperChem Release 5.0 allows a user to replace the standard menu structure of the shipped product with a totally new and custom menu structure. Each custom menu item can have its own button text and can be tied to the execution of an arbitrary script file, either `*.SCR` or `*.TCL`. Since all the conventional menu items of HyperChem Release 5.0 have equivalent script commands such as *menu-file-open*, the custom menus can replicate the standard HyperChem product as well as define essentially any new product that one likes. The custom menus allow a Tabula Rasa, or blank page, on which a new chemistry product can be written.

Interfacing to HyperChem

A principal component of the CDK is the documentation, libraries, and examples that allow you to interface your own codes to HyperChem. Customizing HyperChem, as implied above, means the *internal* execution of script commands brought about by reading them from a simple text file (`*.SCR` or `*.TCL`). Interfacing to HyperChem on the other hand, implies that an *exter-*

nal program (your own) executes the equivalent of script commands by sending them as *messages* to a running copy of HyperChem.



That is, an external program can drive HyperChem from outside by sending it script messages. Interfacing to HyperChem does not imply that you statically link your code together with HyperChem code but rather that you completely control HyperChem from outside and read and write to HyperChem's data structures via external messages.

Dynamic Data Exchange

With most computer operating systems, running programs can send messages to one another. In the Unix environment these messages are often implemented with Pipes or Sockets and the Unix version of HyperChem can accept messages sent by external programs in this fashion. Microsoft Windows and NT include a capability, referred to as Dynamic Data Exchange (DDE), that is an equivalent capability allowing one Windows or NT program to communicate with another by sending it DDE messages.

External Script Messages

In the Microsoft Windows or NT environment a program external to and independent of HyperChem can drive or control HyperChem by sending it DDE messages. HyperChem responds to a complete set of *script messages*, implemented via DDE, that are analogous and essentially identical to the *script commands* discussed above. A script command is text placed in a file while a script message is the corresponding text placed in a message. Thus a HyperChem molecular dynamics trajectory can be initiated by running a script with the text *do-molecular-dynamics* in a `*.scr` or `*.tcl` file or by sending a DDE message containing the same text.

These external DDE messages can be sent to HyperChem from essentially any well-designed Microsoft Windows or NT program. Thus, HyperChem can be driven from a word processor like Microsoft Word, a spreadsheet like

Microsoft Excel, a simple Visual Basic program or a C, C++, or Fortran program.

The HyperChem Application Programming Interface (HAPI)

The CDK includes a dynamic link library (HAPI.DLL) and a static library (HAPI.LIB) that makes it particularly easy to interface your compiled Visual Basic, C, C++, or Fortran program to HyperChem. Instead of using DDE, an external program can just make HAPI calls to HyperChem. These HAPI calls are part of a higher-level interface that is included with the CDK and which abstracts away from operating system and machine dependencies to give you a portable interface to HyperChem. Everything you can do with the lower-level DDE interface you can accomplish with the HyperChem API.

Overview of Chapters

The *CDK Manual* contains the following chapters:

- Chapter 1, this "*Introduction,*" discusses the general nature of the components of the Chemist's Developer Kit and gives an overview of the rest of the chapters.
- Chapter 2, "*Architecture of HyperChem,*" discusses the general architecture of HyperChem including its *Front End - Back End* Architecture, its *Network* Architecture, its *Client-Server* Architecture, and its *Open* Architecture. Each of these architectural aspects is important for the richer understanding of HyperChem that is desirable in order for you to be able to easily customize it or interface your own code to it. Of particular importance for the CDK is an understanding of the Open Architecture of HyperChem.
- Chapter 3, "*Customizing HyperChem,*" describes script commands and the execution of script files, either HyperChem Hcl script files or Tcl/Tk Script files. Hcl script files contain simple sequences of script commands. Tcl/Tk Script files contain normal Tcl/Tk code with embedded Hcl script commands. This chapter, in addition, gives a full description of the custom menu capability.
- Chapter 4, "*HyperChem State Variables,*" describes the concept of HyperChem State Variables (HSV's) in detail. The basic data structures of HyperChem are HSV's. These are registered by HyperChem at instantiation and are made available for reliable Reading and Writing by scripts and by external programs. This chapter provides background for later

chapters which use HSV's in scripts and in the HyperChem Application Programming Interface (HAPI).

- Chapter 5, "*Custom Menus*," describes the concept of custom menus and the syntax of a menu file. A menu file describes a set of menus and the scripts that are executed upon selecting a menu item. This chapter fully defines the use of custom menus in HyperChem.

- Chapter 6, "*Type 1 (Hcl Scripts),*" describes the HyperChem Command Language (Hcl) and scripts based upon it. These scripts are straight-line scripts that access HyperChem data and functionality but include no control structures. Even without the superstructure provided by the control structures of other languages, however, Hcl scripts can accomplish significant tasks on their own.

- Chapter 7, "*Type 2 (Tcl/Tk) Scripts,*" describes a new feature of HyperChem, the inclusion of a very flexible and extensible scripting language referred to as the Tool Command Language (Tcl). It includes an extension called the Toolkit (Tk) that can be used to build additional graphical user interfaces for HyperChem. All Type 1 (Hcl) script commands are fully imbedded in the new Tcl/Tk interpreter.

- Chapter 8, "*DDE Interface to HyperChem,*" describes the low level DDE interface to HyperChem that allows the direct control of HyperChem by programs like Word or Excel. A first example of HyperChem interfacing is given by having these external programs execute simple script messages to affect the visual appearance of the HyperChem screen. Subsequent, more complicated examples, are also described.

- Chapter 9, "*DDE and Visual basic,*" describes how programs written in Visual Basic can be interfaced to HyperChem via DDE. This extends the discussion of the last chapter to a more fully programmable system like Visual Basic. It contrasts with later Visual Basic applications that use the HAPI library.

- Chapter 10, "*The External Tcl/Tk Interface,*" describes an interface to HyperChem where external forms of Tcl and Tk are used to interface to HyperChem. Use of theTcl/Tk interpreter from outside HyperChem allows for certain operations, such as notification, that are not possible with the interpreter imbedded right into HyperChem.

- Chapter 11, "*HAPI Interface to HyperChem,*" describes a higher level library for interfacing to HyperChem for use by Visual Basic, C, C++, and Fortran programs. This HyperChem Application Programming Interface (HAPI) is described along with the calls that it contains. The

HAPI library is a fundamental component of the CDK. It is illustrated with a Visual Basic example.

- Chapter 12, "*Development Using the Windows API,*" describes the development of Windows and NT programs that interface to HyperChem. The type of development presented in this chapter uses the Window's System Developer's Kit (SDK) approach, along with HAPI calls. The SDK approach is a detailed and more fundamental way to develop Windows programs than the approach of the next chapter.

- Chapter 13, "*Development Using the MFC,*" describes the development of Windows and NT programs that interface to HyperChem and use the Microsoft Foundation Classes (MFC). The MFC, in conjunction with C++, enables one to build a Windows program much more quickly but with somewhat less flexibility than with the SDK of the last chapter. Development of interfaces to HyperChem using the MFC are described.

- Chapter 14, "Console C and Fortran Applications," describes the interface between Microsoft "console" applications (having no GUI) and HyperChem. The emphasis here is on taking "legacy" Fortran programs and having HyperChem provide a GUI for them.

- The three Appendices describe the complete set of Hcl script commands, the set of Tcl/Tk script commands, and the details of each HAPI call.

Finally, the HyperChem CD-ROM and the associated installation of HyperChem 5.0 will provide you with a number of examples of the scripts, menu files, and applications either discussed or perhaps not discussed in this CDK manual. However, further material associated with the CDK will be found on Hypercube's WWW site (http://www.hyper.com) and you should regularly check that site for additional help.

Chapter 2
Architecture of HyperChem

Introduction

This chapter contains information on the following architectural features of HyperChem:

- The Front End - Back End Architecture
- The Master - Slave Architecture
- The Open Architecture
- The Client - Server Architecture
- The Network Architecture

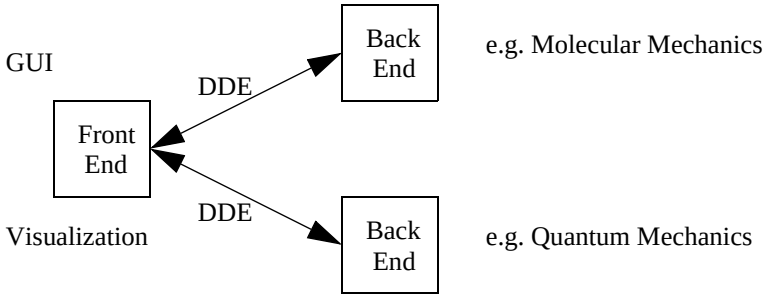
The discussion here is given as general background on how HyperChem is constructed so that you are in a better position to appreciate how and why customization and interfacing become possible. The material of this chapter is not strictly required for what follows in later chapters but should assist you in understanding the CDK.

Since HyperChem, like most large commercial products, has an evolving design and is certainly not the product of a single, totally rational, design process, what is presented here is somewhat of a combination snapshot of both the way it is and the way it is becoming. Nevertheless, what is described here represents Release 5.0 in most regards.

The Front End - Back End Architecture

HyperChem basically consists of one monolithic *Front End* and numerous *Back Ends*. In the Windows and NT environment each of these components has its own icon and is a completely separate program. For example, the green beaker icon represents the HyperChem front end program. The falling red

apple icon represents one of the back end programs - in this case, HyperNewton.



The front end is the program that your interact with. It accepts input from you, via the mouse and the keyboard, which constitute the GUI, and it provides you with visualization services. For example, it may render a drawing of a molecule for you.

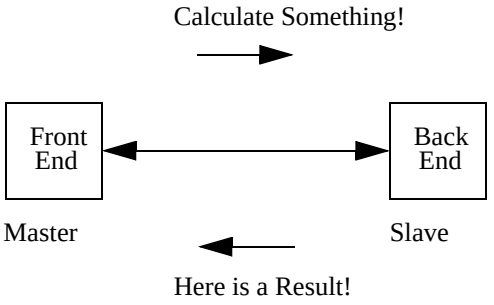
The back ends generally perform the compute intensive computations. HyperChem Release 5.0 comes with 5 back ends - HyperMM+, HyperNewton, HyperEHT, HyperNDO, and HyperGauss. In principal, the back ends compute only the energy of interaction of atoms and the first and second derivatives of these energies. These energetic quantities feed the front end which then computes chemically relevant properties. In practice, the subdivision of labor between a back end program and the HyperChem front end is more complicated than this and depends on the situation.

One of the first things you might wish to consider as an application of the CDK is to implement your own back end to replace one of the HyperChem back ends. For example, you might like to have your own unique force field in replacement of the MM+, Amber, etc. force fields of HyperChem.

In the Windows or NT environment, the HyperChem front end and the various back ends communicate via DDE although one might have thought that they would communicate through files. The HyperChem front end - back end DDE communication, however, is more of a *live link* than would be have been possible via normal file reading and writing.

The Older Master - Slave Architecture

The normal front end - back end relationship in HyperChem Release 5.0 is a master - slave relationship, i.e. the front end is the master over a back end slave.



The back ends do not initiate anything and do only as they are told by the front end. If the front end requires the energy of a molecule, for example, it will know whether a back end slave is idle, it will send the slave a molecule with the instructions, “compute its energy,” and it will then wait for the slave to return the result. The slave, when it is sent a molecule and the instructions for computing some energetic quantity, will do so, returning the result to the front end when it is finished. The slave will then return to an idle state waiting for further instructions.

The Master - Slave Architecture is considered an older architecture by Hypercube. It is slowly being replaced by a newer Client - Server Architecture as described in the next section. However, the existing back ends being shipped by Hypercube as part of Release 5.0 all still use the master - slave relationship. The protocol between a master and a slave in HyperChem is, and has always been, an unpublished proprietary protocol. As such, knowing the explicit details of the protocol is essentially irrelevant to the CDK and to you, the user. It is presented here so that you can understand the basic architecture of HyperChem. The newer client-server protocol described below is the protocol used by the CDK and is the one that you should expect to use in interfacing to HyperChem.

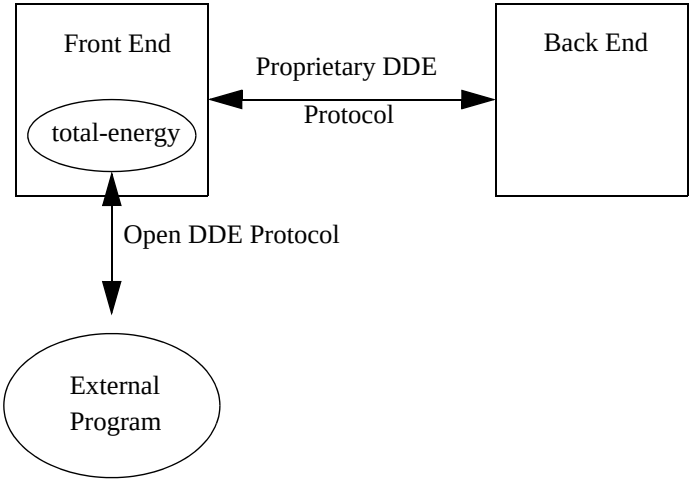
The Open Architecture

HyperChem has an *Open Architecture* in that:

- It can be driven by messages coming from other programs
- Other programs can read and write its internal data structures

The normal operation of HyperChem is via its GUI where you use the mouse and keyboard to operate HyperChem's menus and dialog boxes. HyperChem attempts to allow an external program to operate it in the same way that you do sitting in front of the screen, except that the external program sends script messages (rather than clicking on a key or a mouse button, which a program can't do).

For example, an external program can access an internal variable, such as the total energy, either for reading or writing, by sending a DDE message to the HyperChem front end.

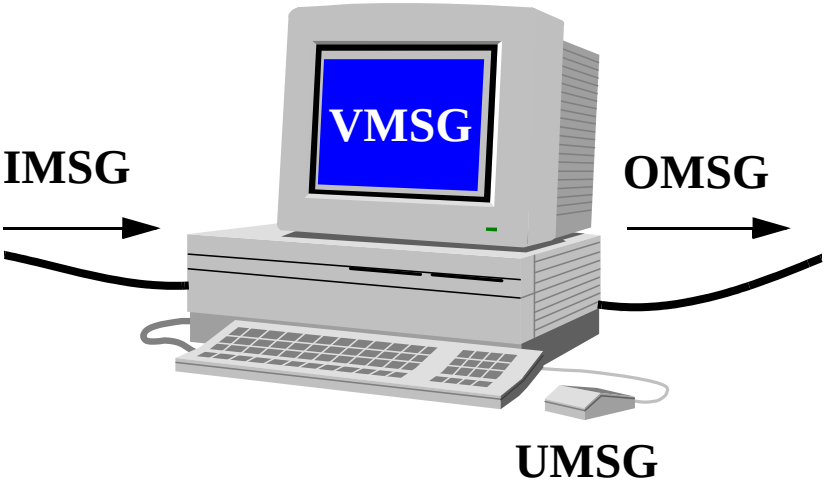


Only front end variables are available to external programs and all information and state held solely by the back end is private to HyperChem. For example, a back end program probably calculated the total energy but conveyed it to the front end where it resides in a front end data structure.

These front end variables that are made available to external programs for reading and writing are referred to as HyperChem State Variables (HSV) and are part of the front end state. The front end HSV's are available for reading and writing at any time whether or not a back end has computed a value. The default value, in this case, is zero.

UMSG and VMSG

Another way of looking at the open architecture of HyperChem is by way of its message interface. In the normal use of HyperChem, via the GUI, a user provides input with mouse clicks and keyboard clicks. We refer to these inputs as user messages. Each constitutes a UMSG. The result of a UMSG is that the internal state of HyperChem changes and a visual change may appear on the screen. We refer to these visual changes as HyperChem having emitted a VMSG. While the user is not explicitly aware of having sent and received these messages, HyperChem does indeed operate this way.



IMSG and OMSG

The open architecture implies that messages equivalent to a UMSG or VMSG can be sent and received by external programs as well as interacting human users. For every UMSG there is expected to be an equivalent input message (IMSG). However, IMSGs are a superset of UMSGs because there are many times a program will want to interface to HyperChem in a way that a human user would not. For example, a program may want to initiate a molecular dynamics trajectory without bringing up a dialog box that a human must respond OK to. The IMSG may or may not generate an output message (OMSG) that augments the VMSG. These IMSGs and OMSGs constitute real data that are conveyed to HyperChem, usually as text strings, by a script or a DDE message.

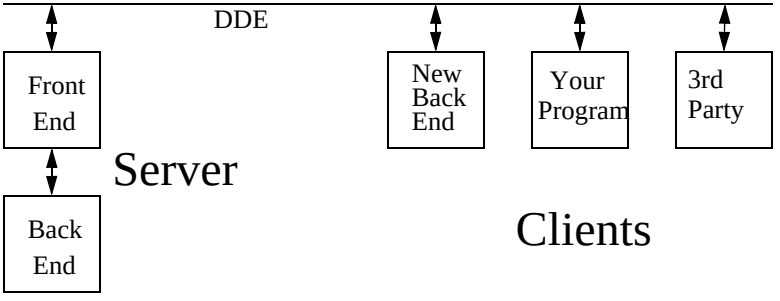
For example, a user clicking with the mouse on the menu item <File/Open> will bring up the File Open dialog box. A program external to HyperChem can do the identical thing by sending an IMSG,

menu-file-open

The resulting OMSG, in this case, is nil but the VMSG is identical to that generated by the UMSG.

The Newer Client - Server Architecture

For a number of reasons it makes sense to move away from the master - slave architecture. In particular, if the services of the HyperChem front end are to be made available to you, other users, and third party programmers, you must be in command of the situation, not HyperChem. Thus HyperChem should act as a universal *server* to you, the *client*.



The HyperChem front end has essentially always acted as a server to external programs as, for example, with ChemPlus, HyperNMR, or programs like Microsoft Excel. The CDK, however, extends and documents this capability, making it possible to even replace Hypercube's proprietary back ends with third party back ends.

These client programs are generally started by the HyperChem front end via a custom menu in HyperChem. When the program begins executing it requests services from HyperChem such as asking it to send a copy of the coordinates of the molecule currently on the screen. It might then compute properties of the molecule and then ask HyperChem to display these properties.

A client program need not reflect just a back end operation but could instead augment the front end GUI or visualization capability of HyperChem. Your program can be of arbitrary design using HyperChem only for functionality

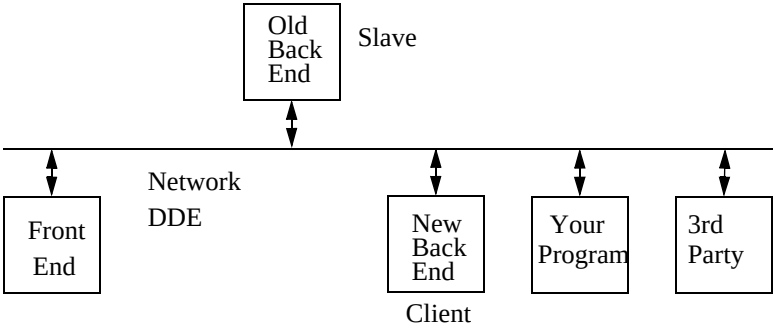
you do not want to reproduce. HyperChem can act as a GUI server, a computational server, or a visualization server all at the same time. The HyperChem back ends are not directly accessible to external programs but only through the HyperChem front end acting as a proxy.

The Network Architecture

As described above, the HyperChem front end, the HyperChem back ends, and third party programs that are DDE compliant, such as Excel or programs built with the Hypercube CDK, can communicate with each other via messages. This communication is initially assumed to occur on a single PC running Windows or NT.

Network DDE

Microsoft, however, has implemented Network DDE, in Windows 95 and NT. Thus it is possible to place any of the various components of a solution onto different PCs as long as they are connected by an appropriate network .



An old back end communicates with the front end by means of Hypercube's proprietary protocol while a new back end uses the open client-server protocol of the CDK.

UNIX

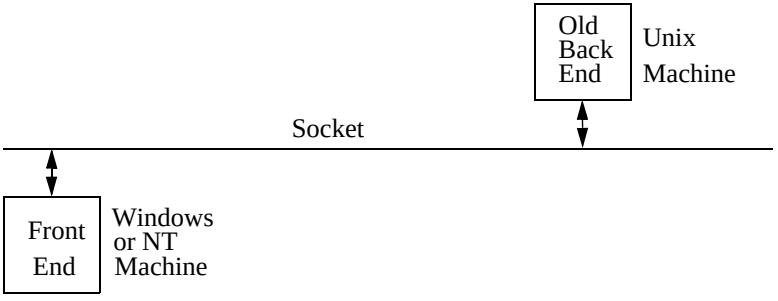
While the following discussion is not strictly relevant to this Windows CDK, a completely analogous CDK and capability is available for Unix versions of HyperChem. In this case DDE is replaced by a Pipe and Socket capability. The Unix versions of HyperChem have a front end - back end architecture as well so that essentially everything we say here applies equally to the Unix

world once DDE messages and communication are replaced by their Unix equivalents. The HyperChem API, on the other hand, is portable across platforms.

Remote Back Ends

Hypercube, Inc. supports the mixing of Windows and NT with Unix in the sense that Remote Unix Back Ends are available for Unix machines from Digital Equipment Corporation, IBM, Silicon Graphics, and SUN. This allows a desktop PC to use a Unix machine to perform compute intensive back end calculations.

These Unix back ends communicate in the older proprietary way with the Windows or NT front end via a proprietary socket protocol. On the Windows or NT side, DDE is replaced by equivalent socket (the WINSOCK standard) messages.



Mixing UNIX and Windows or NT

For completeness it ought to be possible to combine the Windows and NT CDK and the Unix CDK to allow a Unix 3rd party client application to communicate properly with a Windows or NT desktop front end server. Alternatively an NT client might like to use a UNIX machine for HyperChem visualization services. It is not yet possible, however, to mix windows and Unix for these CDK functions. One possible way to obtain this capability (Windows front end HyperChem server and Unix back end client) would be to have a 3rd-party Windows program receive Unix socket messages and translate them into DDE messages for Windows HyperChem.

This and the other network aspects of HyperChem are pointed out here for completeness. This manual and the Windows and NT version of the CDK do

not attempt to describe the equivalent Unix product but relate only to the Windows and NT programs described here.

Chapter 3
Customizing HyperChem

Introduction

This chapter describes how HyperChem can be customized via:

- Scripting
- Custom Menus

These two capabilities allow you to automate many of the computations you perform with HyperChem or to customize HyperChem for your own purposes. This chapter does not describe the interfacing of external programs to HyperChem which, in itself, is a form of customization; that is left for later chapters. Here we focus on how HyperChem can be customized through the process of writing a set of scripts and through your ability, in Release 5.0, to redefine the whole menu structure of the program if you chose to do so.

A Flexible Development Platform

Scripting and custom menus, by themselves, allow you to customize HyperChem in a very large variety of ways. It is possible to, in essence, start from scratch with just a bare Window (no menus) and add only the capabilities you wish. Any menu item can be tied to the execution of an arbitrary script. With Release 5.0 and the CDK, we at Hypercube, Inc. are working towards a Chemical Operating System that allows end users and third-party developers to develop their own products using our tools. With the ability to interface external programs to HyperChem and have them access any of the capabilities of a custom HyperChem, a very flexible chemical development tool becomes available.

What are Scripts?

A script consists of a sequence of individual *script commands* that are held in a script file. Each script command consists of simple text that can be executed to invoke a HyperChem action, to read or write HyperChem data, etc. A script can be created with a text editor such as the Windows Notepad. A script can be executed in HyperChem by simply opening the script with the HyperChem <Script/Open...> menu item. Alternatively, scripts can be created and executed via the Script Editor of ChemPlus.

There are two kinds of scripts. The first consists of a simple straight-line sequence of commands without any control structures (such as for-loops, if-statements, etc.). These are referred to as Type 1 scripts. The second, much richer, type of script has elaborate control structures and is referred to as a Type 2 script. Type 2 scripts contain Type 1 script commands within them.

Type 1 (Hcl) Scripts

The first kind of script, the only kind available prior to Release 5.0, is referred to as a Type 1 Script. These consist of a sequence of text lines of the form,

```
window-color green
```

The above script command, executed within a script, changes the background workspace (window) color to green. Executing this script command is equivalent to using the mouse and the <File/Preferences...> dialog box in the GUI to change the window color. We will often use this simple script command, one of potentially hundreds, to illustrate many of the basic ideas in the CDK.

With Release 5.0 and the CDK, one now refers to these script commands as being part of the HyperChem Command Language (Hcl). Hcl commands in the HyperChem Command Language, minus any possible arguments, are identifiable as a contiguous sequence of words separated by hyphens, e.g. *menu-file-start-log*.

Type 1 scripts are held in `*.SCR` files, i.e they have a default `SCR` file extension.

Type 2 (Tcl/Tk) Scripts

The newer scripts in Release 5.0 are referred to as Type 2 Scripts or as Tcl/Tk Scripts. These scripts use a public domain Tcl/Tk interpreter that is part of HyperChem Release 5.0. The interpreter can interpret Hcl script commands

imbedded in normal Tcl/Tk code. The Tcl/Tk interpreter was developed by John Ousterhout and collaborators at the University of California at Berkeley and at Sun Microsystems. It is described in the following books that may be important to you in becoming an efficient developer of Type 2 Scripts:

- Eric F. Johnson, *Graphical Applications with Tcl & Tk*, 1996, M&T Books, New York, N.Y., ISBN 1-55851-471-6.
- John K. Ousterhout, *Tcl and the Tk Toolkit*, 1994, Addison-Wesley, Reading, Mass., ISBN 0-201-6337-X.
- Brent Welch, *Practical Programming in Tcl and Tk, Prentice* Hall, 1994.

It is also described at the following World Wide Web sites:

- http://www.sunlabs.com:80/research/tcl/
- http://www.sco.com/Technology/tcl/Tcl.html

The Tool Command Language orTcl part of Tcl/Tk describes a very general scripting language that can embed custom scripting operations such as the Hcl script commands. The Tk part of Tcl/Tk describes an extension to Tcl that makes it possible to easily create a graphical user interface (GUI) having menus, dialog boxes, etc. Thus a Type 2 script consists of a sequence of script lines that are a mixture of Tcl/Tk (“tickle”) lines and Hcl (“hickle”) lines.

Type 2 scripts add variables, do- or while-loops, if-statements, and so on to a script, so that a script can now be a very general purpose program having its own GUI, visualization, etc. An simple example of a Type 2 script is:

```
TclOnly
set i 10
while { $i > 0 } {
incr i -1
if { ($i - 2*($i/2)) == 1 } then {
hcExec window-color black } else {
hcExec window-color white }
}
Exit
```

This script causes the background window color to alternate between black and white. The while-loop has the variable i going from 10,9,8... to 1. For even values of i the window color is set to black while for odd values of i it is set to white. The *TclOnly* command indicates that no Tk window is needed. It should be the first command of any simple script that requires no addition to the HyperChem graphical user interface. The *hcExec* (Hyperchem execute) command is always followed by the appropriate Hcl command, which in this case sets the color of the HyperChem workspace window. Note that Tcl/Tk is case sensitive; a command with the wrong case, such as *Tclonly*, would be an invalid command. A Tcl script is normally exited with the *Exit* command.

Type 2 scripts are held in `*.TCL` files, i.e they have a default `TCL` file extension.

The whole while statement above is really a single Tcl/Tk command. The while command contains other embedded Tcl/Tk commands plus embedded Type 1 script commands. The *hcExec* command embeds Hcl commands that can be menu invocations, such as *menu-file-open*, direct commands, such as *do-molecular-dynamics*, or HSV writes, such as *window-color green*. The corresponding *hcQuery* command embeds HSV reads. Thus, the following Tcl script creates a Tk window with a message, called .msg, contained in the window and displaying the coordinates of each of the atoms:

```
message .msg -text [hcQuery "coordinates"]
pack .msg
```

The *pack* command places widgets, such as .msg, in the main Tk window and controls their layout. Tk widgets are normally named to begin with a “dot”.

Custom Menus

With Release 5.0 of HyperChem, you can now redefine each and every menu item to fully customize HyperChem’s menus. When HyperChem is first invoked, it has a standard set of menus as with any other Windows product. However, if one subsequently executes the Hcl script command:

```
load-user-menu custom.mnu
```

then the standard menus are discarded and replaced by a new set of menus defined by a *menu file* which, in the above case, is the file `custom.mnu`. These menu files define the custom text of each menu button (for each of the new custom menus), and the action that is taken when each menu button is

pushed. The custom action for each menu button, as defined in the file, can be any Hcl script command.

For example, to re-implement the <File/New> menu item exactly as it is shipped in the default product, the custom menu file should request the action,

```
menu-file-new
```

when the appropriate button with text “New” on the “File” menu is pushed. A menu file is basically just a description of the text and the keyboard assist for each new menu button plus an associated Hcl script command for the button.

Because any of the button actions (Hcl script commands) could be of the form,

```
read-tcl-script xxx.tcl
```

it is also possible to have any menu button execute a Tcl script. This results in HyperChem being capable of having very generic menus and functional capability. That is, since a Tcl/Tk script can be executed upon pushing any menu button, and since a Tcl/Tk script can, in principle, accomplish any programming task, there is no fundamental limitation to the generality of HyperChem.

The following menu file, for example, creates a version of HyperChem that has a single menu (“Custom”) with a single menu item (“Version”) which does nothing other than inform the user of the current release number when the menu button is pushed,

```
MENU "&Custom"
   ITEM "&Version",query-value version
END
```

New keyboard accelerators are not available for custom menu items. The default HyperChem keyboard accelerators are always active. However, keyboard assists or shortcuts, usable with the Alt key and indicated by the ampersand, are available.

Before continuing with the description of user-defined menus and scripts for customizing HyperChem, we will first elaborate on the concept of HyperChem State Variables (HSV’s). These are fundamental to the whole concept of customization or interfacing.

Chapter 4
HyperChem State Variables

Introduction

This chapter describes a very important feature of HyperChem, the concept of its current state and the variables that represent that state. These HyperChem State Variables (HSV's) define the current state of visualization and computation in HyperChem, e.g. the color of a window or the total energy of a molecule, and can be read and written by scripts or by external programs. these HSV's are registered when HyperChem is invoked and can be reliably queried or modified from then on.

Registering of HSV's

When HyperChem is first invoked, one of the things it does is to register, for reliable import and export, a large number (hundreds) of data structures. In cases these data structures are just simple variables rather than more complicated arrays, lists, etc. By registering these data structures we mean that of all the internal data structures that come and go dynamically in HyperChem, these ones can be requested to be read or written at any time. Any registered variable can be reliably and robustly accessed. If the HSV value is not yet available, HyperChem will issue a simple warning at the attempted access. You needn't worry about writing an HSV value to HyperChem and causing its internal operation to be badly perturbed, apart from the intended effect of having a new value for an HSV.

An Example of an HSV

An example of a HyperChem State Variable is *window-color*. The names for these variables are fixed inside the code of HyperChem and may or may not always be optimally descriptive. The variable *window-color* defines the background of the HyperChem workspace (the area displaying a molecular system). It is commonly black but can be changed via the <File/Preferences...>

menu item to be any one of the 8 basic HyperChem colors. The variable is an enumerated variable of type *Enum*. That is, it is one of a small number of predefined values represented internally in the computer by an integer but in your program code as some member of the set {Black, Blue, Green, Cyan, Red, Violet, Yellow, or White}.

Another example of an HSV with more chemical content is *dipole-moment*. This is a R/W variable also and behaves just as *window-color*. However, we prefer to use *window-color* in many of our examples because it is immediately visually obvious when this variable changes. The consequences of writing a new value for the dipole moment are also a little obscure. In actuality no harm comes from writing an arbitrary value for the dipole moment to HyperChem; it just may not correspond to the correct dipole moment for the molecule on the screen. It will be overwritten when a calculation, such as a wave function calculation, is performed that computes the dipole moment.

Read/Write Nature of HSV's

All HSV's can be classified as Read-Only (R) or as Read-Write (R/W). An example of a variable that is read-only is *selected-atom-count*. HyperChem makes extensive use of "atom selections" and this variable reports the total number of currently selected atoms. While it is possible to select atoms via a script, it makes no sense to write new values of this variable that would contradict the number of atoms that HyperChem determines to be currently selected. This number will be automatically updated by HyperChem if an external program were to select some atoms in HyperChem.

Using HSV's

Here we describe the reading and writing of the simplest type of HSV, a scalar. Later, we will describe the reading and writing of HSV's that have a more complicated structure. We use the simple HSV, *window-color*, as our example.

Writing

An HSV is written by simply giving its new value after the variable, with or without an equal sign. Thus the following are all appropriate Hcl script commands for turning the background screen color to green:

```
window-color green
window-color = green
WInDOw-CoLOR=GReeN
```

Hcl script commands are case insensitive and spacing is ignored except within the HSV name.

Reading

Two equivalent syntactical methods are available for reading HSV's. The first is through a Hcl command, *query-value*, which takes an HSV as an argument:

```
query-value window-color
```

The second and completely equivalent way to ask for the value of an HSV is to simply name the HSV placing a question mark after it, separated by at least one space:

```
window-color ?
```

Either of these procedures returns to the questioner (provided the HyperChem screen is green) the output message (OMSG) string:

```
window-color = Green
```

The output message is returned to an external program, if that is who sent the original query or, by default, to a message box on the screen if the original query came from an internal script. Through the use of an Hcl script command, *omsgs-to-file*, the message string generated by the Hcl script can be placed in a file rather than appear in a message box on the screen. Finally, the returning OMSG string could have been shortened to just "Green" rather than "window-color = Green" if the HSV, *query-response-has-tag*, had been set to *false* prior to issuing the original query.

Notifications

It is possible to make a request to HyperChem that you be sent an OMSG should the value of an HSV change. The message you might eventually receive is identical to that which you could receive by performing a *query-value*. The OMSG is repeatedly sent to you whenever the variable changes it

value. For data structures more complex than simple variables, the notification is sent if any member of the data structure changes. For example, monitoring the window-color is accomplished with the script command,

```
notify-on-update window-color
```

The notification can be cancelled at any later time via the script command

```
cancel-notify window-color
```

Notifications are quite powerful and can be extremely useful. For example, you could ask for a notification of the total-energy during an optimization and easily plot a graph of the optimization with the values sent to you by HyperChem. These notifications are really only meaningful in the context of an external program interfaced to HyperChem rather than in the context of an internal script.

Atom Numbering for HSV's

It is important in using HSV's to understand how HyperChem numbers atoms. Each molecule (connected graph) in the HyperChem workspace has its own number (1, 2,...). The atoms in any molecule are numbered starting at 1 also. Thus a unique id for an atom includes two numbers - the molecule number and the atom number within the molecule (atom-in-molecule number).

Thus if a number is to be used as a unique index for an atom, it has the form, (atom number, molecule number). For example, (2,1) is the second atom in the first molecule. The atom index always comes before the molecule index in conventional HSV usage.

Argument Types for HSV's

The types of arguments that HSV variables have are the following:

Boolean	Yes or no, true or false, 0 or 1.
string	Text (letters, characters, or symbols, in upper- or lowercase, unlimited number of characters). Enclose a string in quotes (" ") if it contains spaces, tabs, or newline characters.
filename	A type of string requiring a DOS filename.
enum	A type of string requiring one of a limited set of possibilities.
int	An integer.

float A floating point (decimal) number. For an angle, the number is in degrees.

An int or float may have limits which are checked. For example, create-atom takes an int that is restricted to the range (1..103) and creates an atom with this atomic number at the origin of the molecular system. Floats may have similar limits. A string may not always need to be enclosed in quotes ("string") but it is safer to do so. Appropriate values for an enum depend on the context, of course.

Kinds of HSV's

A number of different types of HSV's are available. They are classified by how they are assigned values. The simplest are just scalars.

Scalar HSV's

A scalar HSV is one which does not use an index. A simple example is *max-iterations*, the maximum number of allowed iterations in a self-consistent-field (SCF) calculation. The argument is an int in the range (1..32767) and an assignment (write) looks as follows:

```
max-iterations = 100
```

The equal sign is not strictly necessary and white space will do:

```
max-iterations 100
```

The number of arguments for a scalar HSV, while normally one, it not restricted to one. For example the HSV, *dipole-moment-components*, takes three float arguments - the x, y, and z components. It is defined as a scalar since you do not use an array index with it but assign or query all its components simultaneously. It is written or assigned, with a flexible syntax, as follows:

```
dipole-moment-components 1.0 2.0 3.0
dipole-moment-components = 1.0 2.0 3.0
dipole-moment-components = 1.0, 2.0, 3.0
```

That is, the arguments may begin with an equal sign or not, be separated by commas or not, etc. The syntax is flexible but equal signs and commas are suggested as an appropriate convention.

Vector HSV's

A vector HSV is one which takes a single index. An example is the HSV, *alpha-orbital-occupancy*. This variable describes the number of electrons in an alpha (spin up) molecular orbital from an unrestricted Hartree-Fock (UHF) calculation. When the calculation is a restricted Hartree-Fock (RHF) calculation having no beta occupied orbitals different from alpha occupied orbitals then the *alpha-orbital-occupancy* variable describes the total occupancy of alpha and beta electrons in an orbital. Thus the arguments are 0 and 1 for UHF calculations and 0 and 2 for RHF calculations. Thus, for H_2 you can externally set the first (HOMO) orbital to be occupied and the second (LUMO) orbital to also be occupied (corresponding to H_2^{2-}) by the following script commands:

```
alpha-orbital-occupancy(1) 2
alpha-orbital-occupancy(2) 2
```

or, equivalently,

```
alpha-orbital-occupancy(1) = 2
alpha-orbital-occupancy(2) = 2
```

For a standard minimal basis calculation on H_2, the query,

```
alpha-orbital-occupancy ?
```

would return the normal result

```
alpha-orbital-occupancy(1) = 2
alpha-orbital-occupancy(2) = 0
```

That is, even though a query such as

```
alpha-orbital-occupancy(1) ?
```

is perfectly valid, it is possible to query for all relevant indices (1..2) at once, as we have done above. It is, of course, also possible to use the alternative syntax,

```
query-value alpha-orbital-occupancy(1)
```

Array HSV's

An array HSV is defined to be one which takes two indices. These two indices are *always* the atom index and the molecule index (iat, imol). If one thinks of this combination as a single unique index, then an array is just a vector where the index is a unique atom number. A simple example is the mass of an atom which is represented by the HSV, *atom-mass*. The atomic masses of H_2 would be assigned as follows,

atom-mass(1,1) = 1.008

atom-mass(2,1) = 1.008

That is, the first and second atoms of molecule one are assigned. Another example is the Cartesian coordinates of the atoms. This is an array with 3 arguments analogous to the dipole moment components above. The coordinates of H_2, again, could be assigned as follows,

coordinates (1,1) = 0.0, 0.0, 0.0

coordinates (2,1) = 0.0, 0.0, 0.74

An alternative to this is to assign all coordinates at once,

coordinates = 0 0 0 0 0 0.74

Note again that there is flexibility in using commas or equal signs or not using them.

A Finite State Machine View of HyperChem

HyperChem has the characteristics of a Finite State Machine (FSM). By this it is meant that HyperChem has a finite set of internal states; an input received while in a particular state causes a transition to a new state with the consequent emission of an output. The set of states and the set of inputs/outputs associated with the set of possible state transitions fully characterizes an FSM.

In the HyperChem case, the inputs are messages and the outputs are messages. these messages are of four kinds (two input and two output):

- User message (UMSG) - an input message coming from the user using the mouse of keyboard.

- Visual message (VMSG) - an output message representing a visual change in the screen.
- Input message (IMSG) - an input message coming from a script or from a Dynamic Data Exchange (DDE) message.
- Output message (OMSG) - an output message going to a script directed target or to an external receiver of DDE messages.

The normal interactive use of HyperChem has UMSGs as input and VMSGs as output. That is, when you click the mouse, something changes on the screen. It is possible to drive HyperChem with IMSGs rather than a mouse or keyboard. With an IMSG for input, one obtains possibly both a VMSG and an OMSG for output. That is, an IMSG, in addition to causing a visual change on the screen may result in an OMSG being sent to a receiver associated with the sender of the IMSG.

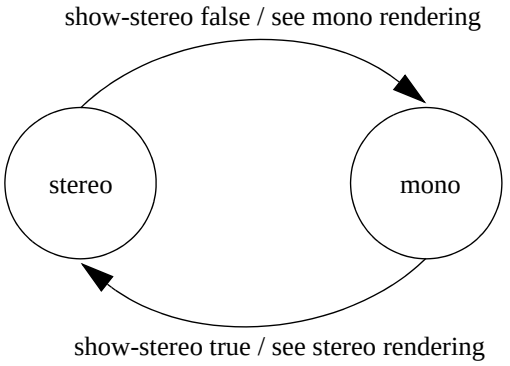
IMSGs are meant to be a superset of UMSGs. It is intended that anything you can do with the mouse you can do with an IMSG. In addition, IMSGs will trigger actions and state changes that are impossible by direct interaction. The target for OMSGs depends on the source of the IMSG and on the redirection of OMSGS by previous IMSGs. These generic concepts have two current implementations. One is implemented by the Hcl scripting language and the other is implemented by dynamic data exchange (DDE).

As an example of the state machine aspects of HyperChem, consider two states of HyperChem shown below as circles. The state is whether a stick rendering is shown on the screen as a normal image or as a stereographic image. Equivalently, the two states are described by an internal HyperChem State Variable (HSV) called *show-stereo*. when this variable is false, a normal (mono) image is displayed but when this variable is true, a double (stereo) image is displayed. The transitions between the two states are represented as arrows. the transition is triggered by an <input> and results in an <output> and each transition (arrow) is labelled:

transition: <input>/<output>

The input in this case is an IMSG of type “show-stereo”, the name of the HSV, with an argument (part of the data of the message) of either “true” or “false”. The output message is simply a VMSG that changes the screen display. No OMSG is emitted in this case.

.



An HSV Server View of HyperChem

The HyperChem State Machine acts as a universal HSV Server in a Client-Server architecture. This is consistent with HyperChem attempting to be a core component of any chemical computation. Other programs, scripts, and third-party processes are clients that HyperChem serves. Consider a third-party application such as one that you would potentially write. If you want the molecular coordinates to perform a calculation with, you must make a request to HyperChem for the coordinates rather than expect HyperChem to send them on its own. HyperChem is sitting in a loop servicing requests like this as they come in. When it receives the request from you for coordinates, it sends them and then looks for another request. Some of these requests come from a user interacting with a mouse and some come from you and your program requesting further data, requesting the ability to access certain HyperChem functionality, or requesting a change in HyperChem's internal state via the setting of a HyperChem State Variable (HSV). These server requests are what we have called UMSGs or IMSGs above. A UMSG comes from an interacting user clicking on a mouse or the keyboard while an IMSG is an internal request in a script or an external request coming in a DDE message.

Further information on HSV's is contained in Chapter 6, Appendix A, or the HyperChem Reference Manual.

Chapter 5

Custom Menus

Introduction

This chapter describes in detail the customizing of HyperChem via your ability to add menu items to the <Script> menu plus your ability to redefine completely the menu structure with a menu (`*.mnu`) file.

Script Menu Items

There has been the ability in earlier releases of HyperChem to add menu items to the <Script> menu. This capability is still there, with two slightly different ways of invoking it (using the "third" menu item as an example):

(1) change-user-menuitem 3, "Sample", "sample.scr"

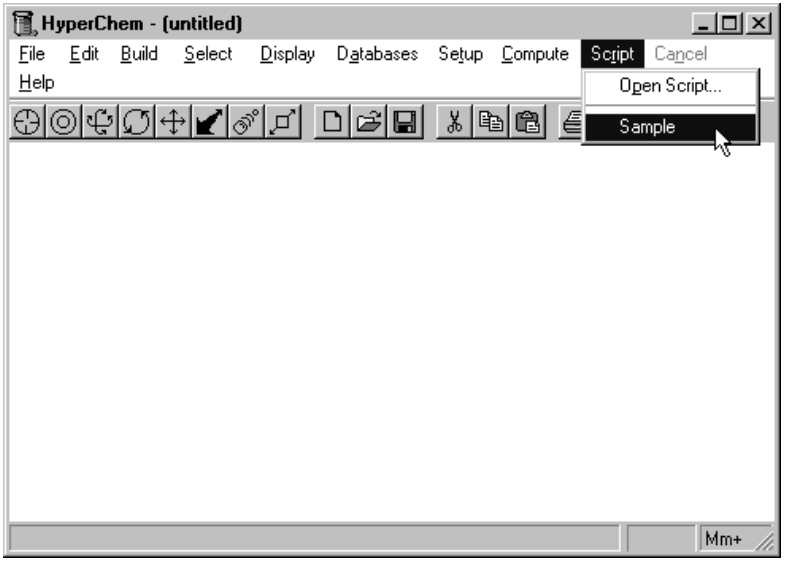
script-menu-enabled(3) = true

(2) script-menu-caption(3) = "Sample"

script-menu-command(3) = "read-script sample.scr"

script-menu-enabled(3) = true

Executing the above script, `Example1.scr` from the HyperChem CD-ROM, illustrates this behavior. For simplicity, the `sample.scr` file contains only the single Hcl script command, *window-color=green*, which gives a quick visual feedback of the effect of the new menu item. The new menu item is shown below:



Up to ten new menu items, with vector indices 1..10, are available under the <Script> menu. Each new menu item shows up in the order associated with the vector index which was assigned to it, such as the “3” above. Menu items that have not been assigned a caption do not show up.

Menu Files

What is new in Release 5.0 of HyperChem is that, in addition to adding new menu items in the <Script> menu, all the menu items can now be designed from scratch and instead of just executing a Hcl script any menu item can now execute a much more generic Tcl/Tk script.

This new functionality comes about via the three new Hcl script commands,

load-user-menu <menu file (*.mnu)>

load-default-menu

switch-to-user-menu

The first of these loads and switches HyperChem to a set of custom menus defined in a menu file. The remaining two switch back and forth between the default *hard-wired* menus and the user *custom* menus, provided they have been defined earlier by the first script command of the above three.

A menu file need not have the `*.mnu` extension since it simply a text file but it is convenient for menu files to be recognized with their own extension. These menu files define the left-to-right and top-to-bottom structure of a complete set of menus via sequential text lines of the form,

MENU <text-string>

ITEM <text-string>, <Hcl-script-command>

ITEM ...

END

The text string is the text that you see on the menu or menu item, including the naming of a possible keyboard assist via the ampersand, "&", being placed prior to the character that is to be used with the Alt key, as per the Microsoft Windows standard. If a menu has no subset of items but is its own menuitem and can be activated by clicking on it alone, it is defined in the form,

MENUITEM <text-string>, <Hcl-script-command>

Cascading menus are not supported. The menu file that corresponds to the default set of hard-wired menus is on the HyperChem CD-ROM as `default.mnu`. It looks as follows:

```
;
; 1996 (c) Hypercube, Inc.
; User Customizable Menu
;
MENU "&File"
   ITEM "&New\tCtrl+N",menu-file-new
   ITEM "&Open...\tCtrl+O",menu-file-open
   ITEM "&Merge...",menu-file-merge
   ITEM "&Save\tCtrl+S",menu-file-save
   ITEM "Save &As...\tCtrl+A",menu-file-save-as
   SEPARATOR
   ITEM "S&tart Log...",menu-file-start-log
   ITEM "Stop Lo&g",menu-file-stop-log
   ITEM "&Log Comments...",menu-file-log-comments
   SEPARATOR
   ITEM "&Import...",menu-file-import
   ITEM "&Export...", menu-file-export
   SEPARATOR
   ITEM "&Print...\tCtrl+P", menu-file-print
   SEPARATOR
   ITEM "Pr&eferences...",menu-file-preferences
```

```
   SEPARATOR
   ITEM "E&xit",menu-file-exit
END
MENU"&Edit"
   ITEM "&Clear\tDelete",menu-edit-clear
   ITEM "C&ut\tCtrl+X",menu-edit-cut
   ITEM "C&opy\tCtrl+C",menu-edit-copy
   ITEM "Copy ISIS S&ketch",menu-edit-copy-isis-sketch
   ITEM "&Paste\tCtrl+V",menu-edit-paste
   SEPARATOR
   ITEM "Cop&y Image\tF9",menu-edit-copy-image
   SEPARATOR
   ITEM "&Invert",menu-edit-invert
   ITEM "&Reflect",menu-edit-reflect
   SEPARATOR
   ITEM "Rotat&e...",menu-edit-rotate
   ITEM "&Translate...",menu-edit-translate
   ITEM "&Zoom...",menu-edit-zoom
   ITEM "Z C&lip...",menu-edit-z-clip
   SEPARATOR
   ITEM "&Align Viewer...",menu-edit-align-viewer
   ITEM "Align &Molecules...",menu-edit-align-molecules
   SEPARATOR
   ITEM "Set &Bond Length...",menu-edit-set-bond-length
   ITEM "Set Bon&d Angle...",menu-edit-set-bond-angle
   ITEM "Set Bo&nd Torsion...",menu-edit-set-bond-torsion
END
MENU"&Build"
   ITEM "&Explicit Hydrogens",menu-build-explicit-hydrogens
   ITEM "&Default Element...",menu-build-default-element
   ITEM "&Add Hydrogens",menu-build-add-hydrogens
   ITEM "&Model Build",menu-build-model-build
   SEPARATOR
   ITEM "Allow &Ions",menu-build-allow-ions
   ITEM "&United Atoms",menu-build-united-atoms
   ITEM "A&ll Atoms",menu-build-all-atoms
   SEPARATOR
   ITEM "Calculate T&ypes",menu-build-calculate-types
   ITEM "Compile Type &Rules",menu-build-compile-type-rules
   SEPARATOR
   ITEM "&Set Atom Type...",menu-build-set-atom-type
   ITEM "Set &Mass...",menu-build-set-mass
   ITEM "Set C&harge...",menu-build-set-charge
```

```
   ITEM "&Constrain Geometry...",menu-build-constrain-geometry
   ITEM "Constrain &Bond Length...",menu-build-constrain-bond-length
   ITEM "Constrain Bond An&gle...",menu-build-constrain-bond-angle
   ITEM "Constrain Bond &Torsion...",menu-build-constrain-bond-
torsion
END
MENU"&Select"
   ITEM "&Atoms",menu-select-atoms
   ITEM "&Residues",menu-select-residues
   ITEM "&Molecules",menu-select-molecules
   SEPARATOR
   ITEM "M&ultiple Selections",menu-select-multiple-selections
   ITEM "&Select Sphere",      menu-select-select-sphere
   SEPARATOR
   ITEM "S&elect All",menu-select-select-all
   ITEM "&Complement Selection",menu-select-complement-selection
   ITEM "Se&lect...",menu-select-select
   ITEM "&Name Selection...",menu-select-name-selection
   SEPARATOR
   ITEM "E&xtend Ring",menu-select-extend-ring
   ITEM "Ex&tend Side Chain",menu-select-extend-side-chain
   ITEM "Exten&d to sp3",menu-select-extend-to-sp3
   ITEM "Select Bac&kbone",menu-select-select-backbone
END
MENU"&Display"
   ITEM "&Scale to Fit\tSpace",menu-display-scale-to-fit
   ITEM "&Overlay",menu-display-overlay
   SEPARATOR
   ITEM "Show &All",menu-display-show-all
   ITEM "Sho&w Selection Only",menu-display-show-selection-only
   ITEM "&Hide Selection",menu-display-hide-selection
   SEPARATOR
   ITEM "&Rendering..."menu-display-rendering
   ITEM "Last Renderin&g\tF2"menu-display-last-rendering
   SEPARATOR
   ITEM "Show &Isosurface\tF3"menu-display-isosurface
   ITEM "Isosur&face...\tF4"menu-isosurface-control
   SEPARATOR
   ITEM "Show H&ydrogens",menu-display-show-hydrogens
   ITEM "Show Periodic &Box",menu-display-show-periodic-box
   ITEM "Show &Multiple Bonds",menu-display-show-multiple-bonds
   ITEM "Show Hy&drogen Bonds",menu-display-show-hydrogen-bonds
   ITEM "Recomp&ute H Bonds",menu-display-recompute-h-bonds
```

```
   ITEM "Show Inertial A&xes",menu-display-show-inertial-axes
   ITEM "Show Dipole Momen&t",menu-display-show-dipole-moment
   SEPARATOR
   ITEM "&Labels...",menu-display-labels
   ITEM "&Color...",menu-display-color
   ITEM "&Element Color...",menu-display-element-color
END
MENU"D&atabases"
   ITEM "&Amino Acids...",menu-databases-amino-acids
   ITEM "&Make Zwitterion",menu-databases-make-zwitterion
   ITEM "&Remove Ionic Ends"menu-databases-remove-ionic-ends
   SEPARATOR
   ITEM "&Nucleic Acids...",menu-databases-nucleic-acids
   ITEM "Add &Counter Ions",menu-databases-add-counter-ions
   SEPARATOR
   ITEM "M&utate...",menu-databases-mutate
END
MENU "Se&tup"
   ITEM "&Molecular Mechanics...",menu-setup-molecular-mechanics
   ITEM "&Semi-empirical...",menu-setup-semi-empirical
   ITEM "&Ab Initio...",menu-setup-ab-initio
   SEPARATOR
   ITEM "&Periodic Box...",menu-setup-periodic-box
   ITEM "&Restraints...",menu-setup-restraints
   ITEM "Set &Velocity...",menu-setup-set-velocity
   SEPARATOR
   ITEM "S&elect Parameter Set...",menu-setup-select-parameter-set
   ITEM "C&ompile Parameter File",menu-setup-compile-parameter-file
   SEPARATOR
   ITEM "&Reaction Map...", menu-setup-reaction-map
END
MENU    "&Compute"
   ITEM "&Single Point",menu-compute-single-point
   ITEM "&Geometry Optimization...",menu-compute-geometry-
optimization
   SEPARATOR
   ITEM "&Molecular Dynamics...",menu-compute-molecular-dynamics
   ITEM "&Langevin Dynamics...",menu-compute-langevin-dynamics
   ITEM "Monte &Carlo...",menu-compute-monte-carlo
   SEPARATOR
   ITEM "&Vibrations",menu-compute-vibrations
   ITEM "&Transition State...",menu-compute-transition-state
   SEPARATOR
```

```
   ITEM "&Plot Molecular Properties...", menu-compute-plot-molecular-
properties
   ITEM "&Orbitals...",menu-compute-orbitals
   SEPARATOR
   ITEM "Vi&brational Spectrum...",menu-compute-vibrational-spectrum
   ITEM "&Electronic Spectrum...",menu-compute-electronic-spectrum
END
MENU "Sc&ript"
   ITEM "O&pen Script...",menu-script-open-script
   ITEM "&Compile Script...",menu-script-compile-script
END
MENUITEM "Ca&ncel",menu-cancel
MENU "&Help"
   ITEM "&Index",menu-help-index
   ITEM "&Keyboard",menu-help-keyboard
   ITEM "&Commands",menu-help-commands
   ITEM "&Tools",menu-help-tools
   ITEM "&Scripts \& DDE",menu-help-scripts-&-DDE
   ITEM "&Glossary",menu-help-glossary
   ITEM "&Using Help",menu-help-using-help
   SEPARATOR
   ITEM "&About HyperChem", menu-help-about-hyperchem
END
```

An extension of this menu file, called `chemplus.mnu`, is on the HyperChem CD-ROM to enable users of Release 5 to continue using ChemPlus from menu items. [Although ChemPlus 1.5 continues to work with HyperChem 5.0, the ChemPlus modules cannot be activated by menu items except through a menu file such as described here].

Simple Example

The custom menus can be illustrated by executing the script, `Example2.scr` from the HyperChem CD-ROM:

```
script-menu-caption(1) = "Rotate Menus"

script-menu-command(1) = "load-user-menu rotate.mnu"

script-menu-enabled(1) = true
```

This script puts a new menu item in the <Script> menu. The new menu button has the text, “Rotate Menus” on it. Pushing this menu button will execute a script command, as shown above, that loads a custom set of user menus (`rotate.mnu`) rather than the hard-wired default menus that come with the

product. A set of custom menus is characterized by a menu file (`*.mnu`), which in this case is the set of menus defined in `rotate.mnu` (from the HyperChem CD-ROM):

```
MENU "&File"
    ITEM "&New\tCtrl+N",menu-file-new
    ITEM "&Open...\tCtrl+O",menu-file-open
    ITEM "E&xit",load-default-menu
END
MENU"&Rotate"
    ITEM "&X Axis",read-tcl-script rotatex.tcl
    ITEM "&Y Axis",read-tcl-script rotatey.tcl
    ITEM "&Z Axis",read-tcl-script rotatez.tcl
END
MENUITEM "Ca&ncel",menu-cancel
```

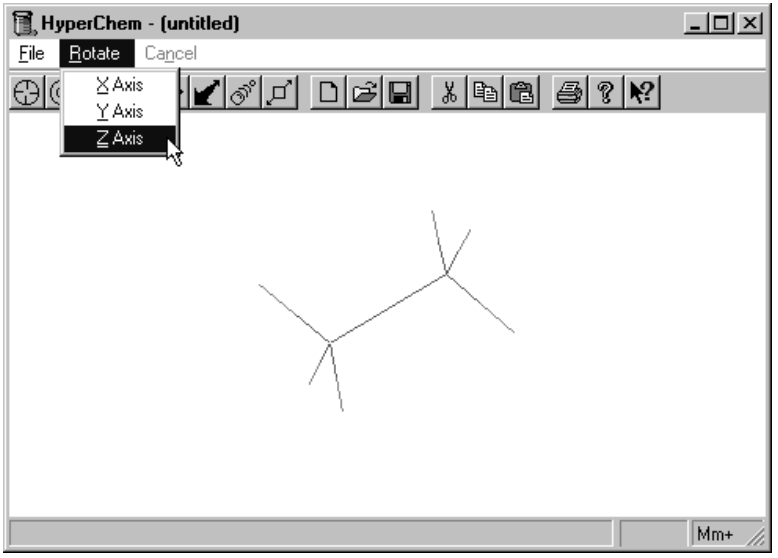
This menu file is characteristic of all menu files in that it simply defines the individual menus and menu items from left to right and top to bottom. The key words MENU, MENUITEM and ITEM introduce a normal menu, a menu without any underlying items, and an individual menu item within a menu. Each menu is defined by the text string label for that menu while each menu item is defined by the text string plus a Hcl script command that is executed when that menu item is chosen. Each text string can be assigned a keyboard assist with the ampersand, “&”.

The way to have this file define menus that are identical to the default hard-wired menus is to define a menu item with a script command that is a *menu invocation* such as,

```
menu-file-new
```

A menu item can be set to execute any Hcl script command. By using the script commands, *read-script* and *read-tcl-script*, any generic Hcl script or Tcl/Tk script stored in a file can be triggered by each custom menu item. In the above case, a Tcl script file is attached to each member of a set of three menu items for continuously rotating the molecular system about the x, y, or z axis. The menu item <File/Exit> has been set to execute a script command that returns to the hard-wired default menus. In this way it is simple to go back and for between the two sets of menus by just clicking on menu items, i.e. by clicking on <Script/Rotate Menus> to obtain the special menus for rotation only and by clicking on <File/Exit> to return to the default menus. Executing

the original `Example2.scr` script and then clicking on <Script/Rotate Menus> gives a version of HyperChem that is shown below.



The rotatez.tcl script, which is included on the HyperChem CD-ROM, rotates the molecule about the z axis and is as follows:

```
TclOnly
hcExec "query-response-has-tag false"
hcExec "cancel-menu = true"
for {set i 0} {$i < 36500} {incr i} {
hcExec "rotate-molecules x 5"
set we_quit [hcQuery cancel-menu]
if { $we_quit == "false" } { break }
update
}
```

This is not the place to describe Tcl/Tk scripts in detail but this script is fairly simple and illustrates many of the general ideas of how universal is the ability to customize HyperChem. The first Tcl command, TclOnly [note that Tcl/Tk is case sensitive] indicates that this script needs no graphical user interface

GUI) or extra windows. By default, without this command, a simple window is put up where any Tk widgets are placed. Since we do not want this window, we indicate that this Tcl/Tk script is a Tcl only script without any extra visual elements.

The HyperChem command language, Hcl, and the HyperChem GUI do not have any capability for continuously rotating a molecule. The GUI allows for manually rotating a molecule and the script command,

rotate-molecules x <angle>

rotates the molecular system about, in this case, the x axis by <angle> degrees as a one-shot, non-continuous motion. To see continuous motion, the <angle> must be small and the script command repeated a large number of sequential times even for a "single" rotation. The way to obtain continuous rotation, of course, is to have a do, while or for-loop of these individual rotation commands. The Tcl code above has a simple "for-loop over i" of 36500 iterations. It remains only to describe the Hcl script commands that are embeded inside Tcl and the handling of the Cancel menu.

The Tcl language embeds all possible Hcl commands into the language through the syntax,

hcExec <Hcl script command> and

hcQuery <name of HSV>

The basic idea here is to place the Hcl script command, *rotate-molecules* (about the z axis by 5 degrees), inside the for-loop as is done in the above script. The remaining complexity of the Tcl script is all associated with the handling of the Cancel button. Without the Cancel button, the rotation would go on forever, or at least for 7200 complete rotations. The trick is to sense whether the Cancel button has been pushed while executing the loop and break out of the loop if this is so. As with all queries of HSV's, there is the possibility of the answer coming back with a tag, such as "cancel-menu=true" or as just the raw value, "true". We first ensure that only raw values are returned so as to simplify the parsing required. We thus execute the Hcl script command,

query-response-has-tag = false

Next, we disable (gray-out) the usual menus and enable (blacken) the Cancel menu via the following script command, just prior to beginning the rotation,

cancel-menu = true

This (Cancel-enabled, Normal-disabled) state of HyperChem can be monitored by enquiring about the HSV, *cancel-menu*. If it is true, then the Cancel button is enabled and it can be clicked upon at any time. If you do click on

this Cancel button, it becomes disabled and the value of the HSV becomes false. The Tcl script looks for this cancel operation each iteration and breaks out of the loop if it occurs. The only other Tcl command that needs comment is the update command. Without this, Tcl would not temporarily release control to Windows and sense the change made to the cancel-menu HSV in HyperChem. It would be busy just executing “its own for-loop code”. When you want an immediate update of a Tk dialog box or a HyperChem HSV, it is best to place an appropriate update command inside a Tcl script.

Further Customization

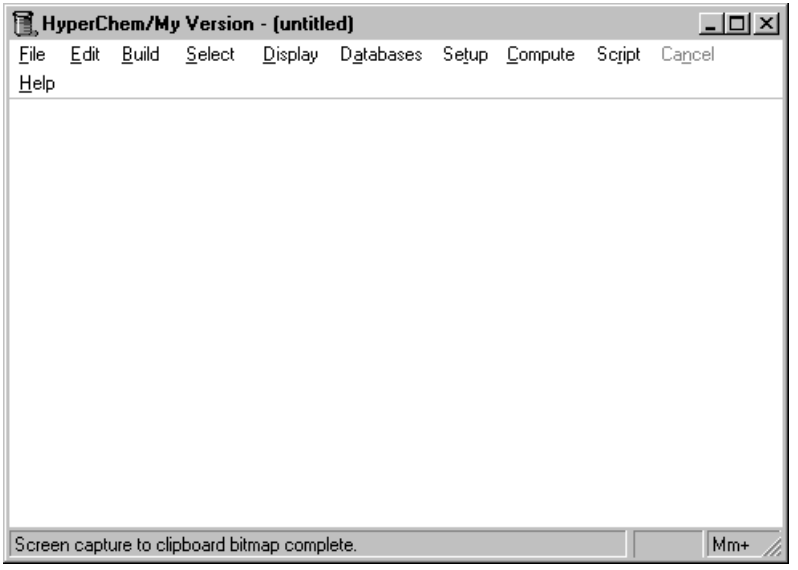
The name of the HyperChem window can be modified to identify specific custom versions of HyperChem. The is very straight forward using the following Hcl script command.

```
custom-title <addendum to HyperChem name>
```

In addition, if so desired, the tool bar can be eliminated using the *hide-toolbar* HSV. Thus, the Hcl script, stored on the HyperChem CD-ROM as `Example3.scr`,

```
custom-title = “My Version”

hide-toolbar = true
```

results in the following window, having a custom name and no toolbar. It is still possible without a toolbar to change the HyperChem cursor tool using the *mouse-mode* HSV.


HyperChem/My Version - (untitled)
File Edit Build Select Display Databases Setup Compute Script Cancel
Help
Screen capture to clipboard bitmap complete.
Mm+

Chapter 6
Type 1 (Hcl) Scripts

Introduction

Type 1 scripts, the simplest scripts, are simple sequences of script commands each of which conform to the *Hyperchem Command Language* (Hcl - pronounced “hickle”). These scripts have been part of HyperChem since its inception but with the addition, in Release 5.0, of Tcl/Tk scripting capability, it is necessary to clarify our terminology somewhat. Thus, what was previously referred to simply as a script command or script message now becomes a Hcl command or Hcl script command. These Hcl script commands can constitute the totality of a script, be imbedded in a Tcl script, or be the content of certain external messages sent to HyperChem by other programs. A script that consists solely of Hcl commands is a Type 1 or Hcl script.

Hcl Script Commands

A Hcl script consists of a sequence of Hcl script commands, each of which is a single line of Ascii text. These Hcl script commands are of four types:

- An HSV read
- An HSV write
- A menu activation
- A direct command

HSV’s

The HyperChem State Variables (HSV’s) have been described in Chapter 4. HyperChem makes these variables available to external programs and to internal scripts, for reading and possibly for writing. They need not be described again here other than for completeness in describing Hcl script commands. Using *dipole-moment* as an example of an HSV, a Hcl script com-

mand can read the value (enquire about the current value that HyperChem maintains) by the Hcl script command,

```
query-value dipole-moment
```

or, alternatively, via the equivalent but slightly different syntax,

```
dipole-moment ?
```

You can write the value (assuming you wish to assign a value of 2.5 Debyes to the dipole moment) via the Hcl script command,

```
dipole-moment 2.5
```

Menu Activations

A menu activation is a replacement for the user clicking with the mouse on a menu item (menu button). Every menu item of HyperChem has an equivalent Hcl script command that accomplishes exactly the same effect as clicking on the menu item. These Hcl script commands all start with "menu-" and then name the particular menu and finally a string representation of the particular menu item, all separated by hyphens. Thus, the equivalent of clicking on the menu item <File/Save As...> is to open a `*.scr` file and execute the Hcl script command,

```
menu-file-save-as
```

All these menu activations are listed in the HyperChem Reference Manual but each can be inferred just by looking at the HyperChem menus and putting a hyphen between every word of the text of the menu item. Thus, if the *Setup* Menu contains the menu button *Select Parameter Set...*, the proper script command is,

```
menu-setup-select-parameter-set
```

One exception, of sorts, to this rule is the "Model Build..." menu item which sometimes reads, "Add H & Model Build..." The script command is always,

```
menu-build-model-build
```

Direct Commands

To open (read in) a file you normally click with the mouse on the menu item <File/Open...> and bring up a dialog box to be filled out with appropriate dialog box values prior to hitting the OK button which initiates the reading in of the file. Suppose you wanted to read in a Protein Data Bank (PDB) file. Then, prior to hitting OK, you need to set the default file type to be `*.ENT` in the dialog box in addition to choosing the name of the appropriate file that you

want to read in. You might like a script to automate these actions but if you used the script command

```
menu-file-open
```

the script would stop with the File Open dialog box sitting on the screen waiting for you to hit the OK button to initiate the reading of the file. This is anything but automation if you have to be there to click on OK! Scripts must go beyond imitating the actions of a user at a keyboard or with mouse in hand.

To solve this problem, a script command is needed that initiates the reading of a file using the current values of the dialog box settings without bringing up the dialog box at all. If different setting are needed than are in the prospective dialog box, these values can be set prior to calling for the opening of the file. The script command *open-file filename* does the job. It is a *direct command* that bypasses a dialog box to get the job done using the current dialog box settings. If you wanted to read, for example, a glucagon PDB file you could simply execute the script,

```
file-format pdb

open-file glucagon.ent
```

This is completely equivalent to invoking the <File/Open...> dialog box, selecting PDB as the file format, filling in the File Name as glucagon.pdb, and hitting OK to dismiss the dialog box and initiate the reading of the file.

Other direct script commands cause actions that have no immediate mapping to a GUI action. The direct script commands consist of the name of the command followed by one or more arguments, each separated by at least one space or a comma:

```
hcl-command-name <argument1>, <argument2>, ...
```

An argument is one of:

Arguments

The types of arguments for variables or commands are the following:

Boolean Yes or no, true or false, 0 or 1.

string Text (letters, characters, or symbols, in upper- or lowercase, unlimited number of characters). Enclose a string in quotes (“ ”) if it contains spaces, tabs, or newline characters.

filename A type of string requiring a DOS filename.

enum A type of string requiring one of a limited set of possibilities.

int An integer.

float A floating point (decimal) number. For an angle, the number is in degrees.

Another example of a direct command is:

```
do-molecular-dynamics
```

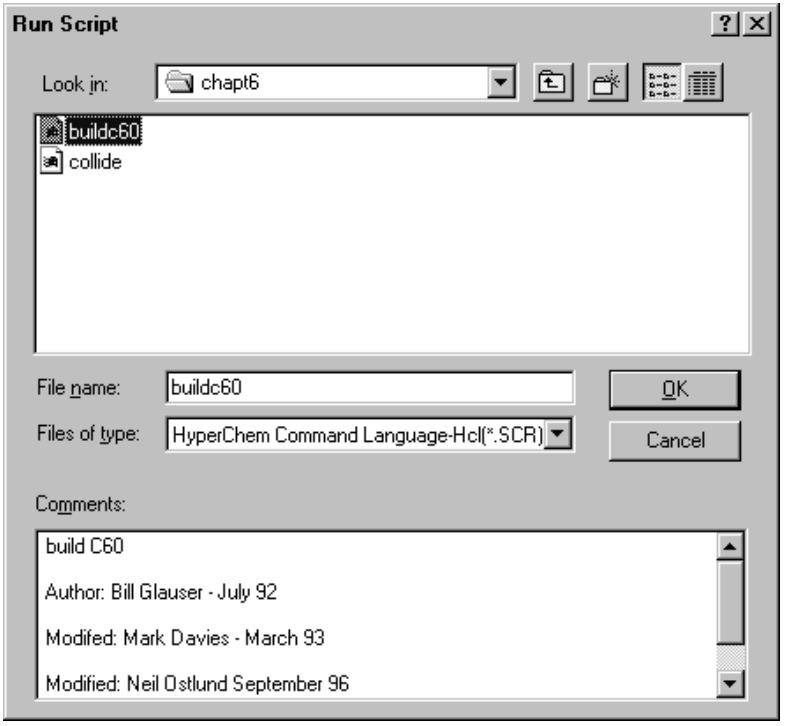
This allows a script to perform molecular dynamics calculations without the manual intervention of a dialog box. Prior to executing this direct command, the script could set all the relevant HSV's for molecular dynamics. This is equivalent to what one is really doing in filling out the molecular dynamics dialog box. For example, the length of the MD trajectory, the dynamics run time, could be set to 100ps via the script command,

```
dynamics-run-time 100
```

Alternatively, one can just accept default values for the relevant HSV's which are just the current values in the dialog box left over from the last time it was invoked. As with almost any situation in HyperChem, reasonable default values are available and are used when you choose not to specify further details.

Script Files

Type 1 scripts are normally stored in files with a default file extension of `*.SCR`. These files consist of nothing more than a sequence of individual Hcl script commands, stored in an Ascii text file with a file name that has the `SCR` file extension. A script is normally executed by opening a `*.SCR` script file with the <Script/Open...> menu item.



A Hcl script file is simply a text file and while it is not essential to use the `*.scr` extension, that is the useful default extension used by HyperChem.

A trivial example of a `*.scr` file is `version.scr`, one that contains the single line,

```
version ?
```

This script file, when opened, queries for the Release Number or Version of HyperChem, in case you are not sure what it is. That is, it requests the value of the HSV, “version”. The result will be reported to you in a message box on the screen.

CHEM.SCR

When HyperChem is started, one of things it does before handing over control to you as a user is to execute a default initialization Hcl script. If HyperChem

finds a script, CHEM.SCR, in its path then it executes that script as a final part of its initialization. If this file does not exist or HyperChem cannot find it in its path, then no such specialized initialization is performed. This allows you to set up a script and place it in CHEM.SCR to customize your copy of HyperChem right from its instantiation, without ever having to explicitly execute a script. If you have your own customization of HyperChem that you like to use on a semi-permanent basis, you should place the relevant script into CHEM.SCR.

Compiled Scripts

A script can be compiled, if desired. The compilation results in a `*.ocr` file. Thus, the following script command compiles `xxx.scr` into `xxx.ocr`,

```
compile-script-file xxx.scr xxx.ocr
```

Compiled scripts can be opened by HyperChem just as text scripts (see the file filter in the <Script/Open...> dialog box).

Recursive Scripts

Scripts files can contain script commands to open other script files. Thus, the script file `a.scr` could contain a script command,

```
read-script b.scr
```

When this second script, `b.scr`, completes, control is returned to the script command in `a.scr` following the above call to `b.scr`.

Script Editor

The companion product to HyperChem, ChemPlus, includes a script editor that makes it easy to create and execute individual Hcl script commands or whole or partial Hcl script files.

Examples

This section begins a description of a number of example Hcl scripts. There are a large number of HSV's and direct script commands contained in HyperChem and learning them all is not a simple matter. One of the best ways is to study example scripts and ultimately to write a number your own scripts. The

examples here will give you some idea of the power of the HyperChem Command Language but its real power ultimately comes in conjunction with the control structures that Tcl can add or the GUI that Tk can add. Hcl scripts are rich in chemistry but it is sometimes difficult to get what you want done easily without the true programming power of variables, do-loops, if-statements, etc.

In addition to this section a good place to start learning Hcl scripts is the `test.scr` script that is explained and described in Chapter 10 of the Reference manual.

Reactive Collision of Two Molecules

This example illustrates a potential process for studying chemical reactions. HyperChem's molecular dynamics (MD) calculations use forces computed by any of the basic computation methods - molecular mechanics, semi-empirical or ab initio quantum mechanics. When the forces are computed by a quantum mechanical method, bond breaking is quite straight-forward and an MD trajectory can describe a reactive or non-reactive collision of two molecular systems. The actual calculation of a rate constant for the chemical reaction is much more complicated and involves performing the collision over and over again with a Boltzman weighted set of initial conditions and averaging over the results. A Tcl script could do this but here we just illustrate how to set up a simple script so that any two molecules on the screen could be set to collide with each other and you can see what happens. To reiterate, this will show you only one of the many possible outcomes of collisions between the two molecules. It nevertheless can be very informative.

This script assumes there are two molecules and two molecules only on the screen. A collision is going to be initiated between a moving molecule 1 and a stationary molecule 2. Molecule 1's center of mass is going to be set to move in the direction of the center of mass of molecule 2.

Assign Target Position

The first thing we are going to do is to make the named selection POINT correspond to molecule 2. POINT is a pre-named selection such that POINT can be referred to by other parts of HyperChem as the center-of-mass of the selection. That is POINT can be both a name for a particular selection as well as the center-of-mass of that selection. We select the atoms of molecule 2 and assign them to POINT. We can select all the atoms of one molecule by having the selection unit be molecules (menu item <Select/Molecules>) and selecting any atom of that molecule.

```
select-none                ; start clean
selection-target molecules ; want to select all of molecule 2
select-atom 1,2            ; select atom 1 of molecule 2 => all atoms
name-selection POINT       ; name the selection POINT
set-velocity POINT 0       ; make sure molecule 2 is stationary
select-none                ; de-select
```

Assign Collision Velocities

The next step is to assign the velocities of the atoms of molecule 1 to be such that the center of mass of molecule 1 is aimed at the center of mass of molecule 2. We do this by selecting all of the atoms of molecule 1 and give them a velocity in the direction of POINT. The scalar value of the assigned velocity here is 200 Angstroms/picosecond. This is rather a high velocity of collision. The center of mass of this second selection, directed at POINT, is the direction of each assigned velocity.

```
select-atom 1,1                    ; select all atoms of molecule 1
set-velocity POINT 200             ; veloc=200 in direction of POINT
select-none                        ; done
```

Wave Function Computation Parameters

Prior to initiating the collision we must set up the parameters for the resulting quantum mechanical computations. In this case we use the CNDO semi-empirical method but it might be any semi-empirical or ab initio method. It is appropriate to try to accelerate the convergence and we do not want to perform a configuration interaction calculation since in that case no forces would be calculated. We will be looking at a classical MD trajectory for the ground state potential energy surface for these two molecules. Configuration Interaction in HyperChem is for exploring excited states. A convergence of 0.01 in usually sufficient and it should definitely converge in less than 100 interations if it converges at all.

The final setting below is to request a UHF rather than an RHF calculation. This is sometimes a controversial subject but RHF calculations do not have the correct asymptotic behavior at long distances when bonds are being broken. UHF calculations sometimes have their own problems but allow a much more generic approach for arbitrary chemical reactions. The UHF calculations usually allow bond-breaking to lead to the correct open-shell behavior of intermediates and products, as compared to RHF calculations.

```
calculation-method=semiempirical    ; faster than ab initio
semi-empirical-method=cndo          ; why not
accelerate-scf-convergence=true     ; fast is good
configuration-interaction=noci      ; no gradients with ci
excited-state=false                 ; could also study lowest state
scf-convergence=0.01                ; 0.1 to 0.0001 ?
max-iterations=100                  ; better converge better than this
uhf=true                            ; essential for bond breaking
```

The Collision

The last step is to initiate the molecular dynamics trajectory correctly. Most of these parameters are not set below but have been set in the dialog box. The important values are the step size and the length of the trajectory. If gradients get very high a smaller step size may be necessary. A larger collision velocity may also require smaller steps. It is essential to set dynamics-restart to be true as this uses the velocities assigned above rather than attempts to equilibrate the velocities, with random numbers, according to the temperature.

```
dynamics-restart=true               ; use velocities we have
do-molecular-dynamics               ; let 'er rip
```

This example can be run any time there are two molecules on the screen. In some instances, it may be necessary to set the charge and multiplicity for charged or open-shell systems. Try it by placing two methane molecules on the screen and see if you can form $H_2 + C_2H_6$ in a collision? The example can be expanded on in a great many ways. For example, one might like to see molecular orbitals change during the collision.

One of the potential problems in watching molecular dynamics of chemical reaction is that of the “standard model” of HyperChem which treats a molecule as a connected graph. Overlapping spheres is the best way of viewing chemical reactions because they display only the position of atoms and not bonds that may no longer exist. Unfortunately, HyperChem uses the molecular graph to speed up the overlapping spheres rendering; it avoids working out the intersection of spheres when there is no “bond”. Thus one may see rendering artifacts sometimes when two atoms are near each other in a “product” when the reactant molecular graph “says” that they are far apart. An interesting script that you might wish to explore would be one that dynamically recomputes the graph to reflect the changing bonds of the reaction. Because of this issue, sometimes one “bonds everything” so that there are no artifacts in a spheres rendering; quantum mechanical calculations, of course, pay no attention to “bonds” but only care about the position of atoms. Bonds are

something to be derived from the results of the such quantum mechanical calculations.

Building and Optimizing C60

The molecule C_{60} continues to be of great interest to chemists. The power of HyperChem's model builder is illustrated by its capability for building the correct 3D structure of the molecule from a simple drawing of its 2D molecular graph. This can be accomplished without any need for a script if you simply draw the molecular graph using a mouse. This example, however, creates C_{60} using a script as an illustration of the general scripting capability and of the process that one uses in a script to create arbitrary molecular structures. In combination, for example, with Tcl and a Tk GUI it would certainly be possible to add to HyperChem, yourself, the template building capabilities that are the only building capabilities that many other molecular modeling programs contain. That is, the HyperChem model builder has a richness that makes possible many other molecule creation procedures.

Setup

The first part of this and any other script should set up the appropriate HSV environment for the script. Here, this includes options for the model builder, for the rendering and for the optimization that comes later. It is particularly important to note that one should draw with explicit hydrogens (hydrogens are not added unless they are explicitly drawn). This means that for intermediate structures used to build up to C_{60}, the model builder will not add inappropriate intermediate hydrogens to the dangling valencies.

```
file-needs-saved no              ; no user intervention - new
menu-file-new                    ; clean slate for new
render-method sticks             ; use sticks for drawing
calculation-method molecular-mechanics       ; for later optimization
molecular-mechanics-method mm+   ; universal method
selection-target atoms           ; select individual atoms
show-multiple-bonds yes          ; lets see aromatic bonds
allow-ions yes                   ; valence of 4 for S
explicit-hydrogens yes           ; build won't add wrong hydrogens
multiple-selections yes          ; going to select group
```

Drawing the First Pair of Atoms

To draw a molecule, one uses *create-atom* to place an atom, of a specified atomic number, down onto the workspace. This script command places all atoms at the origin and the fact that two atoms may have the same coordinates is not particularly important here as their final positions will be chosen by the model builder, which cares here only about the graph (what is bonded to what!) not the arbitrary initial coordinates. If the coordinates were to be important because, say, the model builder was not to be used, then any atom could be translated (translate-selection) to some arbitrary Cartesian coordinates.

Once, for example, the first two atoms are placed onto the workspace, a bond is placed between them with the script command, *set-bond*. The order of the arguments is atom and molecule of the first atom and then atom and molecule of the second atom. In this case, the two atoms are both numbered 1 but in molecules 1 and 2. The two atoms are in separate molecules, *until the bond is drawn*, because they are not part of the same connected molecular graph which the bonds define.

```
;build first bonded pair of atoms
create-atom 6                       ; place C at origin
create-atom 6                       ; place 2nd C at origin
set-bond 1 1  1 2  a                ; create aromatic bond between
menu-build-model-build              ; build 3D structure
```

Finish First Level Pentagon

The following script code then creates the remaining three atoms of the first pentagon, building the structure (applying the model builder) as we go along. It is not really necessary to apply the model builder at every step. Finally the ring closing bond is applied and the atoms are all colored red. The top and bottom pentagons will be colored red just to distinguish them as we rotate the final structure. HyperChem generally uses a “select then operate” algorithm where you first select a subset of atoms of the molecular system, apply some operation to this subset, and then de-select or not for the next operation. The operation here is one of many possible ones that could be applied to a selection. Specifically it is the *color-selection* operation.

```
create-atom 6                       ; create the remaining atoms
set-bond 2 1  1 2  a
```

```
menu-build-model-build
create-atom 6
set-bond 3 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 4 1  1 2  a
menu-build-model-build
set-bond 5 1  1 1  a              ; set ring-closure bond
menu-build-model-build
select-atom 1 1                   ; select each of the five atoms
select-atom 2 1
select-atom 3 1
select-atom 4 1
select-atom 5 1
color-selection red               ; color them red
select-none                       ; de-select everything
```

Build Remaining Layers

The following code is repetitive and just adds the remaining 55 atoms according to the correct graph, building as it goes along.

```
create-atom 6                     ; build second tier
set-bond 1 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 2 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 3 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 4 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 5 1  1 2  a
menu-build-model-build
create-atom 6                     ; build third tier
set-bond 6 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 6 1  1 2  a
menu-build-model-build
```

```
create-atom 6
set-bond 7 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 7 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 8 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 8 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 9 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 9 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 10 1  1 2  a
menu-build-model-build
create-atom 6
set-bond 10 1  1 2  a
menu-build-model-build
set-bond 11 1  19 1  a
set-bond 12 1  14 1  a
set-bond 13 1  16 1  a
set-bond 15 1  18 1  a
set-bond 17 1  20 1  a
create-atom 6                          ; build fourth tier
set-bond 11 1  1 2  a
create-atom 6
set-bond 12 1  1 2  a
create-atom 6
set-bond 13 1  1 2  a
create-atom 6
set-bond 14 1  1 2  a
create-atom 6
set-bond 15 1  1 2  a
create-atom 6
set-bond 16 1  1 2  a
create-atom 6
set-bond 17 1  1 2  a
```

```
create-atom 6
set-bond 18 1  1 2  a
create-atom 6
set-bond 19 1  1 2  a
create-atom 6
set-bond 20 1  1 2  a
set-bond 21 1  22 1  a              ; tier of five pentagons
set-bond 23 1  24 1  a
set-bond 25 1  26 1  a
set-bond 27 1  28 1  a
set-bond 29 1  30 1  a
menu-build-model-build
create-atom 6                       ; fifth tier
set-bond 21 1  1 2  a
create-atom 6
set-bond 22 1  1 2  a
create-atom 6
set-bond 23 1  1 2  a
create-atom 6
set-bond 24 1  1 2  a
create-atom 6
set-bond 25 1  1 2  a
create-atom 6
set-bond 26 1  1 2  a
create-atom 6
set-bond 27 1  1 2  a
create-atom 6
set-bond 28 1  1 2  a
create-atom 6
set-bond 29 1  1 2  a
create-atom 6
set-bond 30 1  1 2  a
menu-build-model-build
set-bond 32 1  34 1  a
set-bond 33 1  36 1  a
set-bond 35 1  38 1  a
set-bond 37 1  40 1  a
set-bond 31 1  39 1  a
create-atom 6
set-bond 31 1  1 2  a
create-atom 6
set-bond 32 1  1 2  a
create-atom 6
```

```
set-bond 33 1  1 2  a
create-atom 6
set-bond 34 1  1 2  a
create-atom 6
set-bond 35 1  1 2  a
create-atom 6
set-bond 36 1  1 2  a
create-atom 6
set-bond 37 1  1 2  a
create-atom 6
set-bond 38 1  1 2  a
create-atom 6
set-bond 39 1  1 2  a
create-atom 6
set-bond 40 1  1 2  a
menu-build-model-build
set-bond 41 1  42 1  a
set-bond 43 1  44 1  a
set-bond 45 1  46 1  a
set-bond 47 1  48 1  a
set-bond 49 1  50 1  a
create-atom 6
set-bond 41 1  1 2  a
set-bond 49 1  51 1  a
create-atom 6
set-bond 42 1  1 2  a
set-bond 44 1  52 1  a
create-atom 6
set-bond 43 1  1 2  a
set-bond 46 1  53 1  a
create-atom 6
set-bond 45 1  1 2  a
set-bond 48 1  54 1  a
create-atom 6
set-bond 47 1  1 2  a
set-bond 50 1  55 1  a
menu-build-model-build
create-atom 6
set-bond 51 1  1 2  a
create-atom 6
set-bond 52 1  1 2  a
create-atom 6
set-bond 53 1  1 2  a
```

```
create-atom 6
set-bond 54 1  1 2  a
create-atom 6
set-bond 55 1  1 2  a
menu-build-model-build
set-bond 56 1  57 1  a
set-bond 57 1  58 1  a
set-bond 58 1  59 1  a
set-bond 59 1  60 1  a
set-bond 60 1  56 1  a
menu-build-model-build
```

Color Bottom and Rotate

We now color the bottom pentagon, switch to a solid rendering and rotate the molecule to visualize the structure. Because Hcl scripts have no control structures we have to repeat the *rotate-molecules* script command to get the effect we want. A Tcl script would use a do-loop.

```
select-atom 56 1
select-atom 57 1
select-atom 58 1
select-atom 59 1
select-atom 60 1
color-selection red
select-none
render-method spheres              ; easier to see 3D shape
rotate-molecules x 5               ; rotate around horizontal
rotate-molecules x 5               ; rotate by 5 degrees at time
rotate-molecules x 5               ; rotate until see whole
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
```

```
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
rotate-molecules x 5
```

Zoom Structure

We now switch back to a sticks structure and zoom in on the molecule for effect. We see a perspective picture as we zoom.

```
render-method sticks
align-viewer z                    ; view molecule along z axis
front-clip 0                      ; don't let it get clipped
show-perspective true             ; pretty perspective view
translate-view 0 0 1              ; translate 1 Angstrom in Z
translate-view 0 0 1              ; repeat according to taste
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
translate-view 0 0 1
```

```
translate-view 0 0 1
translate-view 0 0 -30              ; go back where we were
show-perspective false            ; turn off for other renderings
```

Create an SO_2 Molecule Inside C_{60}

Next we create an SO_2 molecule and place it at the center inside the C_{60} structure. We switch back to a solid rendering and cut away the from of C_{60} so that we can see the SO_2 molecule inside. If the selection unit (target) is molecules then selecting any atom of a molecule selects the whole molecule. The SO_2 is built without affecting the C_{60} by selecting a subset (SO_2) before building. This performs an incremental build (an important capability of the model builder) on only the selection ("select and operate").

```
selection-target molecules            ; select whole molecules
select-atom 1 1                       ; select whole C60
color-selection violet                ; color violet for variety
select-none                           ; OK I'm done
create-atom 16                        ; create SO2 - Sulfur first
create-atom 8                         ; then the Oxygen
set-bond 1 2  1 3  d                  ; double bonds
create-atom 8
set-bond 1 2  1 3  d
select-atom 1 2                       ; select the SO2 for a build
menu-build-model-build                ; incremental build
select-none                           ; have good SO2 now
render-method disks
selection-target atoms
front-clip 54
```

Optimize SO_2 inside Cavity

```
do-optimization
exit-script
```

Other scripts can be found on the HyperChem CD-ROM. You may wish to explore them as a CDK learning tool. To complete this chapter, we catalog all Hcl script commands.

Catalog of HSV's and Direct Script Commands

What follows is an alphabetic listing of all HSV *variables* and all direct *commands*. Each entry contains an indication whether it is a variable or command and for a variable whether it is a Readonly variable or a Read/Write variable.

For variables the next line describes the type of variable it is while for commands the argument list is described. The third line of each entry gives a brief description of the entry. These entries are the result of running the script command, *print-variable-list*, which is the final arbiter of the complete list of script commands. The list of menu activations is left off this list since they can be inferred from HyperChem by just looking at its menus. Alternatively, the menu activations are described in the HyperChem Reference Manual.

abinitio-buffer-size: Variable, Read/Write.
    Type: integer in range (1 .. 32767).
    Two electron integral buffer size.
abinitio-calculate-gradient: Variable, Read/Write.
    Type: boolean.
    Enable Ab Initio gradient calculation (Single Point only).
abinitio-cutoff: Variable, Read/Write.
    Type: float in range (0 .. 1e+010).
    Two electron integral cutoff.
abinitio-d-orbitals: Variable, Read/Write.
    Type: boolean.
    Either five (False) or six (True).
abinitio-direct-scf: Variable, Read/Write.
    Type: boolean.
    Enable Ab Initio Direct SCF calculation.
abinitio-f-orbitals: Variable, Read/Write.
    Type: boolean.
    Either seven (False) or ten (True).
abinitio-integral-format: Variable, Read/Write.
    Type: enum(raffenetti, regular).
    Either regular or raffenetti.
abinitio-integral-path: Variable, Read/Write.
    Type: string.
    Path for storing integrals.
abinitio-mo-initial-guess: Variable, Read/Write.
    Type: enum(core-hamiltonian, projected-huckel, projected-cndo,projected-indo).
    Either core-hamiltonian, projected-huckel, projected-cndo, projected-indo.
abinitio-mp2-correlation-energy: Variable, Read/Write.
    Type: boolean.
    Enable Ab Initio MP2 correlation energy.
abinitio-mp2-frozen-core: Variable, Read/Write.
    Type: boolean.
    Enable Ab Initio MP2 frozen core.
abinitio-scf-convergence: Variable, Read/Write.
    Type: float in range (0 .. 100).
    SCF Convergence for Ab Initio.
abinitio-use-ghost-atoms: Variable, Read/Write.
    Type: boolean.
    Include or ignore ghost atoms.
accelerate-scf-convergence: Variable, Read/Write.
    Type: boolean.
    Whether to use DIIS procedure.
add-amino-acid: Command.
    Arg list: string.
    String-1 gives the name of an amino acid residue to add to the system.

add-nucleic-acid: Command.
Arg list: string.
String-1 names the nucleotide to add to the current system.
align-molecule: Command.
Arg list: .
Align the inertial axes of the molecular system.
align-viewer: Command.
Arg list: .
Align the viewer's line-of-sight with the indicated axis or LINE.
allow-ions: Variable, Read/Write.
Type: boolean.
Whether to allow excess valence on atoms.
alpha-orbital-occupancy: Variable, Read/Write.
Type: vector of float.
(i) Number of electrons in the i-th MO.
alpha-scf-eigenvector: Variable, Read/Write.
Type: vector of float-list.
(i) Coefficients for the i-th MO.
amino-alpha-helix: Command.
Arg list: (void).
Subsequent additions of amino acid residues are to use alpha-helix torsions.
amino-beta-sheet: Command.
Arg list: (void).
Subsequent additions of amino acid residues are to use beta-sheet torsions.
amino-isomer: Variable, Read/Write.
Type: enum(l, d).
Whether amino acids are l or d.
amino-omega: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
The Omega amino acid backbone angle.
amino-phi: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
The Phi amino acid backbone angle.
amino-psi: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
The Psi amino acid backbone angle.
animate-vibrations: Variable, Read/Write.
Type: boolean.
Whether or not to animate vibrations.
append-dynamics-average: Command.
Arg list: string.
Add a named selection to dynamics average gathering.
append-dynamics-graph: Command.
Arg list: string.
Add a named selection to dynamics graph display.
append-omsgs-to-file: Command.
Arg list: string.
String-1 gives the name of a file to which o-msgs are to be appended.
assign-basisset: Command.
Arg list: string.
Assign a basis set to a selection or system.
atom-basisset: Variable, Read/Write.
Type: array of string.
(iat, imol) The basis set of atom iat in molecule imol.
atom-charge: Variable, Read/Write.
Type: array of float.

(iat, imol) The charge of atom iat in molecule imol.
atom-color: Variable, Read/Write.
Type: array of .
(iat, imol) The current color of the atom.
atom-count: Variable, Readonly.
Type: vector of integer.
(imol) The number of atoms in molecule imol.
atom-extra-basisset: Variable, Read/Write.
Type: array of string, float.
(iat, imol) The basis set of atom iat in molecule imol.
atom-info: Variable, Readonly.
Type: (unknown).
Funny composite to support backends.
atom-label-text: Variable, Readonly.
Type: array of string.
(iat, imol) RO. The text of the current atom label.
atom-labels: Variable, Read/Write.
Type: enum(None, Symbol, Name, Number, Type, Charge, Mass, Basis-Set, Chirality).
Label for atoms.
atom-mass: Variable, Read/Write.
Type: array of float.
(iat, imol) The mass of atom iat in molecule imol.
atom-name: Variable, Read/Write.
Type: array of string.
(iat, imol) The name of atom iat in molecule imol.
atom-type: Variable, Read/Write.
Type: array of string.
(iat, imol) The type of atom iat in molecule imol.
atomic-number: Variable, Read/Write.
Type: array of integer.
(iat, imol) The atomic number of atom iat in molecule imol.
atomic-symbol: Variable, Readonly.
Type: array of string.
(iat, imol) The element symbol of the atom.
back-clip: Variable, Read/Write.
Type: float.
Set back clipping plane.
backend-active: Variable, Read/Write.
Type: boolean.
Whether current channel is an active backend.
backend-communications: Variable, Read/Write.
Type: enum(Local, Remote).
Whether to compute on local or remote host.
backend-host-name: Variable, Read/Write.
Type: string.
The name of remote host for backend communications.
backend-process-count: Variable, Read/Write.
Type: integer in range (1 .. 32).
The number of processes to run.
backend-user-id: Variable, Read/Write.
Type: string.
The user id to use on the remote host for backend communications.
backend-user-password: Variable, Read/Write.
Type: string.
The password for user id to use on the remote host for backend communications.
balls-highlighted: Variable, Read/Write.

Type: boolean.
Balls and Balls-and-Cylinders should be highlighted when shaded.
balls-radius-ratio: Variable, Read/Write.
Type: float in range (0 .. 1).
Size of the Balls relative to the maximum value.
balls-shaded: Variable, Read/Write.
Type: boolean.
Balls and Balls-and-Cylinders should be shaded.
basisset-count: Variable, Readonly.
Type: integer.
Number of coefficients required to describe a molecular orbital.
bend-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Results from backend computation.
beta-orbital-occupancy: Variable, Read/Write.
Type: vector of float.
(i) Number of electrons in the i-th MO.
beta-scf-eigenvector: Variable, Read/Write.
Type: vector of float-list.
(i) Coefficients for the i-th MO.
bond-color: Variable, Read/Write.
Type: enum(ByElement, Black, Blue, Green, Cyan, Red, Violet, Yellow, White).
The color used for drawing atoms and bonds.
bond-spacing-display-ratio: Variable, Read/Write.
Type: float in range (0 .. 1).
Bond spacing display ratio.
builder-enforces-stereo: Variable, Read/Write.
Type: boolean.
Whether the model builder implicitly enforces any existing stereochemistry.
calculation-method: Variable, Read/Write.
Type: enum(MolecularMechanics, SemiEmpirical, AbInitio).
Whether molecular mechanics, semi-empirical, or ab initio.
cancel-menu: Variable, Read/Write.
Type: boolean.
Whether the cancel menu is up, or the normal one.
cancel-notify: Command.
Arg list: string.
String-1 names a variable to stop watching.
change-stereochem: Command.
Arg list: integer, integer.
Immediately change the stereochemistry about (iat, imol).
change-user-menuitem: Command.
Arg list: integer, string, string.
Change the text and procedure associated with the specified user MenuItem.
chirality: Variable, Read/Write.
Type: array of string.
(iat, imol) A, R, S, or ?, for achiral, R, S, or unknown chirality.
ci-criterion: Variable, Read/Write.
Type: enum(Energy, Orbital).
One of: energy, orbital.
ci-excitation-energy: Variable, Read/Write.
Type: float in range (0 .. 10000).
When ci-criterion=energy, maximum excitation energy.
ci-occupied-orbitals: Variable, Read/Write.
Type: integer in range (0 .. 32767).

When ci-criterion=orbital, count of occupied orbitals included.
ci-unoccupied-orbitals: Variable, Read/Write.
Type: integer in range (0 .. 32767).
When ci-criterion=orbital, count of unoccupied orbitals included.
clip-cursor: Variable, Read/Write.
Type: float in range (0 .. 1000).
Select Z axis clip cursor tool.
clip-icon-step: Variable, Read/Write.
Type: float in range (0 .. 1000).
Select clip step.
color-element: Command.
Arg list: integer, enum().
Element Int-1 gets color String-2 as its default color.
color-selection: Command.
Arg list: string.
String-1 names a color for the current selection.
compile-script-file: Command.
Arg list: string, string.
Compile file string-1, writing result to string-2
configuration: Variable, Read/Write.
Type: integer.
The current UV configuration of the system.
configuration-interaction: Variable, Read/Write.
Type: enum(NoCI, SinglyExcited, Microstate).
One of: no-ci, singly-excited, microstate.
connectivity-in-pdb-file: Variable, Read/Write.
Type: boolean.
Whether connectivity information is to be included in a PDB file.
constrain-bond-angle: Command.
Arg list: float angle in range (-360 .. 360).
Float-1 gives the angle constraint for the three currently selected atoms.
constrain-bond-down: Command.
Arg list: integer, integer, integer, integer.
Constrain the bond from (iat1, imol1) to (iat2, imo2) to be down.
constrain-bond-length: Command.
Arg list: float in range (0 .. 100).
Float-1 gives the length constraint for the two currently selected atoms.
constrain-bond-torsion: Command.
Arg list: float angle in range (-360 .. 360).
Float-1 gives the torsion constraint for the four currently selected atoms.
constrain-bond-up: Command.
Arg list: integer, integer, integer, integer.
Constrain the bond from (iat1, imol1) to (iat2, imo2) to be up.
constrain-change-stereo: Command.
Arg list: integer, integer.
Constrain atom (iat, imol) to change the current stereochemistry.
constrain-fix-stereo: Command.
Arg list: integer, integer.
Constrain atom (iat, imol) to enforce the current stereochemistry.
constrain-geometry: Command.
Arg list: string.
String-1 describes the geometry constraint around the currently selected atom.
coordinates: Variable, Read/Write.
Type: array of float, float, float.
(iat, imol) The x, y, and z coordinates of atom iat in molecule imol.
coordination: Variable, Readonly.
Type: array of integer.

(iat, imol) The coordination number for the specified atom.
cpk-max-double-buffer-atoms: Variable, Read/Write.
Type: integer in range (0 .. 2147483647).
Maximum number of double buffered atoms in cpk rendering mode.
create-atom: Command.
Arg list: integer in range (0 .. 103).
Create a new atom at the origin with atomic number nAtno.
current-file-name: Variable, Readonly.
Type: string.
The name of the current file.
custom-title: Variable, Read/Write.
Type: string.
Custom Title string, append string to title.
cutoff-inner-radius: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
The distance (in Angstroms) to begin a switched cutoff.
cutoff-outer-radius: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
The distance (in Angstroms) at which nonbonded interactions become zero.
cutoff-type: Variable, Read/Write.
Type: enum(None, Switched, Shifted).
Electrostatic cutoff to apply to molecular mechanics calculations.
cycle-atom-stereo: Command.
Arg list: integer, integer.
Advance the stereo constraint about atom (iat, imol).
cycle-bond-stereo: Command.
Arg list: integer, integer, integer, integer.
Advance the stereo constraint along the bond (iat1, imol1)--(iat2, imol2).
cylinders-color-by-element: Variable, Read/Write.
Type: boolean.
Color Cylinders using element colors.
cylinders-width-ratio: Variable, Read/Write.
Type: float in range (0 .. 1).
Width of the Cylinders relative to the maximum value.
d-orbitals-on-second-row: Variable, Read/Write.
Type: boolean.
Include D orbitals on second row.
declare-integer: Command.
Arg list: string.
Declare a new integer variable.
declare-string: Command.
Arg list: string.
Declare a new integer variable.
default-element: Variable, Read/Write.
Type: integer in range (0 .. 103).
The atomic number of the default element for drawing operations.
delete-atom: Command.
Arg list: integer, integer.
Delete the specified atom.
delete-file: Command.
Arg list: string.
filename to be deleted.
delete-named-selection: Command.
Arg list: string.
Remove the named selection String-1 from the list of named selections.
delete-selected-atoms: Command.
Arg list: (void).

Delete the currently selected atoms.
dipole-moment: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Dipole moment.
dipole-moment-components: Variable, Read/Write.
Type: float, float, float.
Dipole moment components.
do-langevin-dynamics: Command.
Arg list: (void).
Perform a Langevin dynamics computation on the system.
do-molecular-dynamics: Command.
Arg list: (void).
Perform a molecular dynamics computation on the system.
do-monte-carlo: Command.
Arg list: (void).
Perform a Monte Carlo computation on the system.
do-optimization: Command.
Arg list: (void).
Perform a structure optimization on the system.
do-qm-calculation: Variable, Read/Write.
Type: boolean.
For single-point QM calculations, whether to re-compute wave function.
do-qm-graph: Variable, Read/Write.
Type: boolean.
For single-point QM calculations, to graph some data.
do-qm-isosurface: Variable, Read/Write.
Type: boolean.
For single-point QM calculations, to generate iso-surface of results.
do-single-point: Command.
Arg list: (void).
Perform a single-point computation on the system.
do-vibrational-analysis: Command.
Arg list: (void).
Perform a vibrational analysis computation on the system.
dot-surface-angle: Variable, Read/Write.
Type: float angle in range (-90 .. 90).
Dot surface angle.
double-buffered-display: Variable, Read/Write.
Type: boolean.
Whether display operations are double-buffered.
dynamics-average-period: Variable, Read/Write.
Type: integer in range (1 .. 32767).
Computation results from dynamics run.
dynamics-bath-relaxation-time: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Bath relaxation time for dynamics.
dynamics-collection-period: Variable, Read/Write.
Type: integer in range (1 .. 32767).
Dynamics data collection interval.
dynamics-constant-temp: Variable, Read/Write.
Type: boolean.
Whether to keep temperature fixed at dynamics-simulation-temp.
dynamics-cool-time: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Time taken to change from dynamics-simulation-temp to dynamics-final-temp.
dynamics-final-temp: Variable, Read/Write.
Type: float in range (0 .. 1e+010).

Temperature to cool back to when annealing.
dynamics-friction-coefficient: Variable, Read/Write.
Type: float in range (0 .. 1000000).
Friction coefficient for Langevin dynamics.
dynamics-heat-time: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Time taken to change from dynamics-starting-temp -> dynamics-simulation-temp.
dynamics-info-elapsed-time: Variable, Readonly.
Type: float in range (0 .. 1e+010).
Elapsed time in dynamics run.
dynamics-info-kinetic-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Computation results from dynamics run.
dynamics-info-last-update: Variable, Readonly.
Type: boolean.
Last update from dynamics run.
dynamics-info-potential-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Computation results from dynamics run.
dynamics-info-temperature: Variable, Readonly.
Type: float in range (0 .. 1e+010).
Computation results from dynamics run.
dynamics-info-total-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Computation results from dynamics run.
dynamics-playback: Variable, Read/Write.
Type: enum(none, playback, record).
Playback a recorded dynamics run.
dynamics-playback-end: Variable, Read/Write.
Type: integer in range (0 .. 32767).
End playback of recorded dynamics run.
dynamics-playback-period: Variable, Read/Write.
Type: integer in range (1 .. 32767).
Dynamics playback interval.
dynamics-playback-start: Variable, Read/Write.
Type: integer in range (0 .. 32767).
Start playback of recorded dynamics run.
dynamics-restart: Variable, Read/Write.
Type: boolean.
Use saved velocities.
dynamics-run-time: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Total integration time at dynamics-simulation-temp.
dynamics-seed: Variable, Read/Write.
Type: integer in range (-2147483648 .. 2147483647).
Seed for dynamics initialization random number generator.
dynamics-simulation-temp: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
High temperature for the dynamics run.
dynamics-snapshot-filename: Variable, Read/Write.
Type: string.
Name file of to store dynamics run.
dynamics-snapshot-period: Variable, Read/Write.
Type: integer in range (1 .. 32767).
Set recording interval of dynamics run.
dynamics-starting-temp: Variable, Read/Write.
Type: float in range (0 .. 1e+010).

Starting temperature for the dynamics run.
dynamics-temp-step: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Step size (K) by which temperature is changed.
dynamics-time-step: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Size of the step for integration.
error: Variable, Read/Write.
Type: string.
The current error.
errors-are-not-omsgs: Command.
Arg list: (void).
Specifies that error messages are to appear in message boxes.
errors-are-omsgs: Command.
Arg list: (void).
Specifies that error messages should be treated like o-msgs.
estatic-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Results from backend computation.
excited-state: Variable, Read/Write.
Type: boolean.
False for lowest state, true for next-lowest state.
execute-client: Command.
Arg list: string.
Run a client application.
execute-hyperchem-client: Command.
Arg list: string.
Run a client application. App can reliably connect to instance of Hyper-Chem.
execute-string: Command.
Arg list: string.
Execute the string variable as a script.
exit-script: Command.
Arg list: (void).
Exit the current script.
explicit-hydrogens: Variable, Read/Write.
Type: boolean.
Whether hydrogens are to be drawn explicitly.
export-dipole: Variable, Read/Write.
Type: boolean.
Whether or not to export dipole moment data to .EXT file.
export-ir: Variable, Read/Write.
Type: boolean.
Whether or not to export IR data to .EXT file.
export-orbitals: Variable, Read/Write.
Type: boolean.
Whether or not to export orbital data to .EXT file.
export-property-file: Command.
Arg list: string.
Writes properties to the named file.
export-uv: Variable, Read/Write.
Type: boolean.
Whether or not to export UV data to .EXT file.
factory-settings: Command.
Arg list: (void).
Reset chem to its out-of-the-box state.
file-diff-message: Command.
Arg list: string, string, string, string.

Compare file1 to file2; if they are the same say string3, else say string4.
file-format: Variable, Read/Write.
Type: string.
The molecule file format.
file-needs-saved: Variable, Read/Write.
Type: boolean.
Whether the current system needs to be saved.
front-clip: Variable, Read/Write.
Type: float.
Set front clipping plane.
global-inhibit-redisplay: Variable, Readonly.
Type: boolean.
Whether redisplay of the system is inhibited (readonly)
graph-beta: Variable, Read/Write.
Type: boolean.
If true and UHF, graph beta-spin orbitals instead of alpha.
graph-contour-increment: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
Increment between contour lines.
graph-contour-increment-other: Variable, Read/Write.
Type: boolean.
Whether to use graph-increment-other (true) or use defaults (false).
graph-contour-levels: Variable, Read/Write.
Type: integer in range (1 .. 32767).
The number of contour levels to plot.
graph-contour-start: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
Value for first contour line.
graph-contour-start-other: Variable, Read/Write.
Type: boolean.
Whether to use graph-contour-start (true) or use defaults (false).
graph-data-row: Variable, Readonly.
Type: vector of float-list.
(i) The values on the i-th row of graph data.
graph-data-type: Variable, Read/Write.
Type: enum(electrostatic, charge-density, orbital, orbital-squared, spin-density).
The type of wavefunction data to plot.
graph-horizontal-grid-size: Variable, Read/Write.
Type: integer in range (2 .. 8192).
Number of data grid points for plotting in the horizontal direction.
graph-orbital-offset: Variable, Read/Write.
Type: integer in range (0 .. +Inf).
Display orbital offset.
graph-orbital-selection-type: Variable, Read/Write.
Type: enum(lumo-plus, homo-minus, orbital-number).
Display orbital type.
graph-plane-offset: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
Offset along viewer's Z axis of the plane of the data to plot.
graph-vertical-grid-size: Variable, Read/Write.
Type: integer in range (2 .. 8192).
Number of data grid points for plotting in the vertical direction.
grid-max-value: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
The isosurface maximum grid value.
grid-min-value: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).

The isosurface minimum grid value.
hbond-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Results from backend computation.
heat-of-formation: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Heat of formation.
help: Command.
Arg list: string.
Give help on topic String-1.
hide-errors: Variable, Read/Write.
Type: boolean.
Whether to display error messages on the screen (channel specific).
hide-messages: Variable, Read/Write.
Type: boolean.
Whether to display MESSAGE value on the screen.
hide-toolbar: Variable, Read/Write.
Type: boolean.
Command to toggle the toolbar.
hide-warnings: Variable, Read/Write.
Type: boolean.
Whether to display warning messages on the screen (channel specific).
huckel-constant: Variable, Read/Write.
Type: float in range (0 .. 10).
Extended Huckel constant.
huckel-scaling-factor: Variable, Read/Write.
Type: float in range (0 .. 100000).
Extended Huckel scaling factor.
huckel-weighted: Variable, Read/Write.
Type: boolean.
Extended Huckel weighting factor.
hydrogens-in-pdb-file: Variable, Read/Write.
Type: boolean.
Should Hydrogens be written into a PDB file?
ignore-script-errors: Variable, Read/Write.
Type: boolean.
Whether script errors should be ignored, otherwise offer to abort on errors.
image-color: Variable, Read/Write.
Type: boolean.
Save image in color?
image-destination-clipboard: Variable, Read/Write.
Type: boolean.
Save image to the Windows clipboard?
image-destination-file: Variable, Read/Write.
Type: boolean.
Save image to a file?
image-file-bitmap: Variable, Read/Write.
Type: boolean.
Save in bitmap format?
image-file-bitmapRGB: Variable, Read/Write.
Type: boolean.
Save in bitmap-RGB format?
image-file-metafile: Variable, Read/Write.
Type: boolean.
Save in Windows metafile format?
image-include-cursor: Variable, Read/Write.
Type: boolean.

Include cursor in image?
image-source-window: Variable, Read/Write.
Type: enum(TopLevel, Workspace, HyperChem, FullScreen).
Extent of image to capture.
import-dipole: Variable, Read/Write.
Type: boolean.
Whether or not to import dipole moment data from .EXT file.
import-ir: Variable, Read/Write.
Type: boolean.
Whether or not to import IR data from .EXT file.
import-orbitals: Variable, Read/Write.
Type: boolean.
Whether or not to import orbital data from .EXT file.
import-property-file: Command.
Arg list: string.
Reads properties from the named file.
import-uv: Variable, Read/Write.
Type: boolean.
Whether or not to import UV data from .EXT file.
info-access: Variable, Readonly.
Type: string.
(RO) Access (R, W, RW) for info-variable-target.
info-enum-id-of: Variable, Readonly.
Type: string.
(RO) Binary id of info-enum-target value for info-variable-target.
info-enum-list: Variable, Readonly.
Type: string.
(RO) If enumerated type, list of enumerated values for info-variable-target.
info-factory-setting: Variable, Readonly.
Type: string.
(RO) factory setting value for info-variable-target.
info-id-of: Variable, Readonly.
Type: integer.
(RO) Binary id of info-variable-target.
info-type-of: Variable, Readonly.
Type: string.
(RO) Type of info-variable-target.
info-type-of-element: Variable, Readonly.
Type: string.
(RO) If info-type-of is array or vector, type of elements.
info-variable-target: Variable, Read/Write.
Type: string.
Variable for which info is required.
inhibit-redisplay: Variable, Read/Write.
Type: boolean.
Whether redisplay of the system is inhibited by current channel.
ir-animate-amplitude: Variable, Read/Write.
Type: float in range (0 .. 10).
The distance in angstroms to move the fastest atom during vib animations(0..10)
ir-animate-cycles: Variable, Read/Write.
Type: integer in range (0 .. +Inf).
The number of cycles (length of time) to animate. 0 means forever.
ir-animate-steps: Variable, Read/Write.
Type: integer in range (3 .. +Inf).
The number of steps to use in animating vibrations (3 -- BIG)
ir-band-count: Variable, Read/Write.

Type: integer.
Number of ir bands.
ir-frequency: Variable, Read/Write.
Type: vector of float.
(i) Frequency of the i-th IR band.
ir-intensity: Variable, Read/Write.
Type: vector of float.
(i) Intensity of the i-th IR band.
ir-intensity-components: Variable, Read/Write.
Type: vector of float, float, float.
(i) Intensity components (x,y, and z) of the i-th IR band.
ir-normal-mode: Variable, Read/Write.
Type: vector of float-list.
(i) Normal node. This is a vector holding x, y, and z for each atom.
is-extended-hydrogen: Variable, Readonly.
Type: array of boolean.
(iat, imol) RO. Is the atom an extended hydrogen?
is-ring-atom: Variable, Readonly.
Type: array of boolean.
(iat, imol) RO. Is the atom in a ring?
isosurface-grid-step-size: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
The isosurface grid stepsize.
isosurface-hide-molecule: Variable, Read/Write.
Type: boolean.
Show only the isosurfaces.
isosurface-map-function: Variable, Read/Write.
Type: boolean.
Display a mapped function isosurface.
isosurface-map-function-display-legend: Variable, Read/Write.
Type: boolean.
Display the isosurface mapped function range legend.
isosurface-map-function-range: Variable, Read/Write.
Type: float, float.
Set the isosurface mapped function range.
isosurface-mesh-quality: Variable, Read/Write.
Type: enum(coarse, medium, fine).
Use coarse, medium or fine grid settings.
isosurface-render-method: Variable, Read/Write.
Type: enum(wire-mesh, Jorgensen-Salem, lines, flat-surface, shaded-surface, Gouraud-shaded-surface, translucent-surface).
The method used to render the isosurfaces.
isosurface-threshold: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
The isosurface threshold value?
isosurface-transparency-level: Variable, Read/Write.
Type: float in range (0 .. 1).
The isosurface level of transparency (0 = opaque, 1 = transparent)
isosurface-x-min: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
The smallest x coordinate of the grid data.
isosurface-x-nodes: Variable, Read/Write.
Type: integer in range (2 .. 128).
The number of isosurface x nodes.
isosurface-y-min: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
The smallest y coordinate of the grid data.
isosurface-y-nodes: Variable, Read/Write.

Type: integer in range (2 .. 128).
The number of isosurface y nodes.
isosurface-z-min: Variable, Read/Write.
Type: float in range (-1e+010 .. 1e+010).
The smallest z coordinate of the grid data.
isosurface-z-nodes: Variable, Read/Write.
Type: integer in range (2 .. 128).
The number of isosurface z nodes.
keep-atom-charges: Variable, Read/Write.
Type: boolean.
Keep atom charges when switch calculation methods.
load-default-menu: Command.
Arg list: (void).
Load the Hyperchem default menu.
load-user-menu: Command.
Arg list: string.
Load user customized menu.
log-comment: Command.
Arg list: string.
Write String-1 into the current logfile.
max-iterations: Variable, Read/Write.
Type: integer in range (1 .. 32767).
Maximum number of SCF iterations.
mechanics-data: Variable, Readonly.
Type: (unknown).
Funny composite to support backends.
mechanics-dielectric: Variable, Read/Write.
Type: enum(Constant, DistanceDependent), enum(Constant, DistanceDependent), enum(Script One), enum().
The method for calculating dielectric permittivity.
mechanics-dielectric-scale-factor: Variable, Read/Write.
Type: float, float, float, float.
Constant to multiply distance-dependent dielectric by.
mechanics-electrostatic-scale-factor: Variable, Read/Write.
Type: float, float, float, float.
Scale factor for 1-4 dielectric interactions.
mechanics-info: Variable, Readonly.
Type: (unknown).
Funny composite to support backends.
mechanics-mmp-electrostatics: Variable, Read/Write.
Type: .
The type of electrostatic interaction to use in MM+ calculations.
mechanics-print-level: Variable, Read/Write.
Type: integer in range (0 .. 9).
Print level for molecular mechanics.
mechanics-van-der-waals-scale-factor: Variable, Read/Write.
Type: float, float, float, float.
Scale factor for 1-4 van der Waals interactions.
merge-file: Command.
Arg list: string.
String-1 names the molecule file to be merged with the current system.
message: Variable, Read/Write.
Type: string.
string1 is an output message.
molecular-mechanics-method: Variable, Read/Write.
Type: enum(mm+, amber, bio+, opls).
The type of molecular mechanics method to perform.
molecule-count: Variable, Readonly.

Type: integer.
The number of molecules in the system.
moments-of-inertia: Variable, Readonly.
Type: boolean.
Return the moments of inertia of selected system.
monte-carlo-cool-steps: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Steps taken to change from dynamics-simulation-temp -> dynamics-final-temp.
monte-carlo-heat-steps: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Steps taken to change from dynamics-starting-temp -> dynamics-simulation-temp.
monte-carlo-info-acceptance-ratio: Variable, Readonly.
Type: float in range (0 .. 1).
Computation result from Monte Carlo run.
monte-carlo-max-delta: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Maximum allowed size of the displacement step in Angstroms.
monte-carlo-run-steps: Variable, Read/Write.
Type: float in range (0 .. 1e+010).
Total number of steps at dynamics-simulation-temp.
mouse-mode: Variable, Read/Write.
Type: enum(Drawing, Selecting, Rotating, ZRotating, Translating, ZTranslating, Zooming, Clipping).
The function of the cursor in the drawing area.
mp2-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
MP2 energy.
multiple-selections: Variable, Read/Write.
Type: boolean.
Allow multiple selections.
multiplicity: Variable, Read/Write.
Type: integer in range (1 .. 6).
Spin multiplicity.
mutate-residue: Command.
Arg list: string.
Change selected residue into String-1.
name-selection: Command.
Arg list: string.
Name the current selection String-1.
named-selection-count: Variable, Readonly.
Type: integer.
The number of named selections.
named-selection-name: Variable, Readonly.
Type: vector of string.
(i) The name of the i-th named selection.
named-selection-value: Variable, Readonly.
Type: vector of float.
(i) The value of the i-th named selection (bond length, angle, etc).
negatives-color: Variable, Read/Write.
Type: enum(ByElement, Black, Blue, Green, Cyan, Red, Violet, Yellow, White).
The color of the negatives.
neighbors: Variable, Readonly.
Type: array of (unknown).
(iat, imol) The neighbor list for the specified atom.
no-source-refs-in-errors: Command.

Arg list: (void).
Controls presentation of filename, line number in error messages.
non-standard-pdb-names: Variable, Read/Write.
Type: boolean.
If true, then look for left-justified pdb element names .
nonbond-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Results from backend computation.
notify-on-update: Command.
Arg list: string.
String-1 gives name of variable whose value-changes are desired.
notify-with-text: Variable, Read/Write.
Type: boolean.
For DDE channels only. Are notifications to be text (otherwise binary).
nucleic-a-form: Command.
Arg list: (void).
Subsequent additions of nucleotides will use a-form torsion angles.
nucleic-alpha: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA alpha backbone torsion.
nucleic-b-form: Command.
Arg list: (void).
Subsequent additions of nucleotides will use b-form torsion angles.
nucleic-backwards: Variable, Read/Write.
Type: boolean.
Build backward DNA.
nucleic-beta: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA beta backbone torsion.
nucleic-chi: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA chi backbone torsion.
nucleic-delta: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA delta backbone torsion.
nucleic-double-strand: Variable, Read/Write.
Type: boolean.
Build double strand DNA.
nucleic-epsilon: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA epsilon backbone torsion.
nucleic-gamma: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA gamma backbone torsion.
nucleic-sugar-pucker: Variable, Read/Write.
Type: enum(2-endo, 3-endo).
Select sugar pucker of DNA.
nucleic-z-form: Command.
Arg list: (void).
Subsequent additions of nucleotides will use z-form torsion angles.
nucleic-zeta: Variable, Read/Write.
Type: float angle in range (-360 .. 360).
DNA zeta backbone torsion.
omsg-file: Variable, Read/Write.
Type: string.
String is file to append omsgs.
omsgs-not-to-file: Command.

Arg list: (void).
Directs that o-msgs are not to be written to a file, but to a messagebox.
omsgs-to-file: Command.
Arg list: string.
String-1 gives the name of the new file to which o-msgs are to be written.
one-line-arrays: Variable, Read/Write.
Type: boolean.
Whether to emit arrays all on one line.
open-file: Command.
Arg list: string.
String-1 names the molecule file to be opened.
optim-algorithm: Variable, Read/Write.
Type: enum(SteepestDescents, FletcherReeves, PolakRibiere, Newton-Raphson, EigenvectorFollow).
The algorithm to use for structure optimizations.
optim-converged: Variable, Readonly.
Type: boolean.
Whether optimization has converged.
optim-convergence: Variable, Read/Write.
Type: float in range (0 .. 100).
Optimization gradient convergence.
optim-max-cycles: Variable, Read/Write.
Type: integer in range (1 .. +Inf).
Maximum number of optimization steps.
orbital-count: Variable, Read/Write.
Type: integer.
Number of molecular orbitals available.
parameter-set-changed: Variable, Read/Write.
Type: boolean.
Toggles the state of the backend parameters.
path: Variable, Read/Write.
Type: string.
Current directory.
pause-for: Command.
Arg list: integer in range (0 .. 32767).
HyperChem pauses for Int-1 seconds.
periodic-boundaries: Variable, Read/Write.
Type: boolean.
Whether to use periodic boundary conditions.
periodic-box-size: Variable, Readonly.
Type: (unknown).
Return the size of the periodic box.
pop-no-value: Command.
Arg list: string.
variable: pop stack, don't restore value.
pop-value: Command.
Arg list: string.
variable: Restore pushed value.
positives-color: Variable, Read/Write.
Type: enum(ByElement, Black, Blue, Green, Cyan, Red, Violet, Yellow, White).
The color of the positives.
print: Command.
Arg list: (void).
Print to default printer.
print-variable-list: Command.
Arg list: string.
Write a summary of the state variables to file String-1

printer-background-white: Variable, Read/Write.
Type: boolean.
Force printer background color to white.
push: Command.
Arg list: string.
variable: Push copy of current value onto stack.
quantum-print-level: Variable, Read/Write.
Type: integer in range (0 .. 9).
Print level for quantum mechanics.
quantum-total-charge: Variable, Read/Write.
Type: integer in range (-32768 .. 32767).
The total charge of the quantum mechanical system.
query-response-has-tag: Variable, Read/Write.
Type: boolean.
Do HSV responses have identifying tags?
query-value: Command.
Arg list: .
String-1 names the state variable whose value should be emitted as an o-msg.
read-binary-script: Command.
Arg list: string.
String-1 names the compiled script file that should be read.
read-script: Command.
Arg list: string.
String-1 names the file that should be read as a command script.
read-tcl-script: Command.
Arg list: string.
String-1 names the file that should be read and executed as a Tcl/Tk script.
remove-all-stereo-constraints: Command.
Arg list: (void).
Remove all stereo constraints.
remove-stereo-constraint: Command.
Arg list: integer, integer.
Remove any stereo constraints from atom (iat, imol)
render-method: Variable, Read/Write.
Type: enum(sticks, balls, balls-and-cylinders, spheres, dots, sticks-and-dots).
How the system is to be displayed.
reorder-selections: Variable, Read/Write.
Type: boolean.
Should atoms in selections be reordered to make proper angles, etc.?
request: Command.
Arg list: string.
Displays String-1 in a modeless dialog until the user click OK.
residue-charge: Variable, Readonly.
Type: array of float.
(ires, imol) The net charge on the residue.
residue-coordinates: Variable, Readonly.
Type: array of float, float, float.
(ires, imol) The center-of-mass for the residue.
residue-count: Variable, Readonly.
Type: vector of integer.
(imol) The number of residues in molecule imol.
residue-label-text: Variable, Readonly.
Type: array of string.
(ires, imol) Text of the label for the residue.
residue-labels: Variable, Read/Write.

Type: enum(None, Name, Sequence, NameSequence).
Label for residues.
residue-name: Variable, Readonly.
Type: array of string.
(ires, imol) The name of the residue.
restraint: Command.
Arg list: string, float, float.
(selection-name, value, force-constant)
restraint-tether: Command.
Arg list: .
(selection-name, [POINT|x,y,z], force-constant)
revert-element-colors: Command.
Arg list: (void).
Use default color scheme for displaying atoms.
rms-gradient: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
Results from backend computation.
rotate-molecules: Command.
Arg list: .
rotate-molecules (axis, angle)
rotate-viewer: Command.
Arg list: .
rotate-viewer (axis, angle)
scf-atom-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
SCF atom energy.
scf-binding-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
SCF binding energy.
scf-convergence: Variable, Read/Write.
Type: float in range (0 .. 100).
Convergence required for QM SCF computations.
scf-core-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
SCF core energy.
scf-electronic-energy: Variable, Readonly.
Type: float in range (-1e+010 .. 1e+010).
SCF electronic energy.
scf-orbital-energy: Variable, Read/Write.
Type: vector of float.
(i) Eigenvalues of the Fock matrix.
screen-refresh-period: Variable, Read/Write.
Type: integer in range (1 .. 32767).
How frequently to update system on the screen.
script-menu-caption: Variable, Read/Write.
Type: vector of string.
Caption for menu button.
script-menu-checked: Variable, Read/Write.
Type: vector of boolean.
If checked.
script-menu-command: Variable, Read/Write.
Type: vector of string.
Command for menu button.
script-menu-enabled: Variable, Read/Write.
Type: vector of boolean.
If greyed.
script-menu-help-file: Variable, Read/Write.

Type: vector of string.
Help file for menu item.
script-menu-help-id: Variable, Read/Write.
Type: vector of integer.
Context id for help on button.
script-menu-in-use: Variable, Read/Write.
Type: vector of boolean.
Is somebody claiming this menu item?
script-menu-message: Variable, Read/Write.
Type: vector of string.
Status message.
script-refs-in-errors: Variable, Read/Write.
Type: boolean.
Whether to include script file line numbers in errors.
select-atom: Command.
Arg list: integer, integer.
Selects atom int-1 in molecule int2. Honors selection-target.
select-name: Command.
Arg list: string.
String-1 specifies the name of a selection to become the current selection.
select-none: Command.
Arg list: (void).
Unselect all atoms.
select-residue: Command.
Arg list: integer, integer.
Selects residue int-1 in molecule int-2, disregarding selection-target.
select-sphere: Variable, Read/Write.
Type: boolean.
Whether double-button dragging selects in a sphere or in a rectangle.
selected-atom: Variable, Readonly.
Type: vector of integer, integer.
(i) The atom and molecule indices of the i-th selected atom.
selected-atom-count: Variable, Readonly.
Type: integer.
The number of selected atoms.
selection-color: Variable, Read/Write.
Type: enum(ThickLine, Black, Blue, Green, Cyan, Red, Violet, Yellow, White).
How to display the selection.
selection-target: Variable, Read/Write.
Type: enum(Molecules, Residues, Atoms).
The type of target for selection operations.
selection-value: Variable, Readonly.
Type: float.
The value of the current selection (bond length, angle, etc).
semi-empirical-method: Variable, Read/Write.
Type: enum(ExtendedHuckel, CNDO, INDO, MINDO3, MNDO, AM1, PM3, ZINDO1, ZINDOS).
The type of semi-empirical computation to perform.
serial-number: Variable, Readonly.
Type: string.
The serial number of this copy of HyperChem, read-only.
set-atom-charge: Command.
Arg list: float in range (-100 .. 100).
Float-1 provides the charge for the currently selected atom(s).
set-atom-type: Command.
Arg list: string.
String-1 provides the type for the currently selected atom(s).

set-bond: Command.
Arg list: integer, integer, integer, integer, enum().
Set bond between (iat1, imol1) and (iat2, imol2) to be bond type.
set-bond-angle: Command.
Arg list: float angle in range (0 .. 180).
Set the bond angle for the current selection.
set-bond-length: Command.
Arg list: float in range (0 .. 3200).
Set the bond length for the current selection.
set-bond-torsion: Command.
Arg list: float angle in range (-360 .. 360).
Set the torsion angle for the current selection.
set-velocity: Command.
Arg list: .
Set the velocity for the selected atoms.
show-axes: Variable, Read/Write.
Type: boolean.
Whether to display inertial axes.
show-dipoles: Variable, Read/Write.
Type: boolean.
Whether to display dipole.
show-hydrogen-bonds: Variable, Read/Write.
Type: boolean.
Whether hydrogen bonds are displayed.
show-hydrogens: Variable, Read/Write.
Type: boolean.
Whether Hydrogens are displayed.
show-isosurface: Variable, Read/Write.
Type: boolean.
Whether an isosurface should be displayed, if one is available.
show-multiple-bonds: Variable, Read/Write.
Type: boolean.
Whether multiple bonds are drawn with multiple lines.
show-periodic-box: Variable, Read/Write.
Type: boolean.
Whether the periodic box is displayed when it exists.
show-perspective: Variable, Read/Write.
Type: boolean.
Whether the system should be displayed in perspective.
show-ribbons: Variable, Read/Write.
Type: boolean.
Whether the system should be displayed with ribbons.
show-stereo: Variable, Read/Write.
Type: boolean.
Whether the system should be displayed as a stereo pair.
show-stereochem-wedges: Variable, Read/Write.
Type: boolean.
Whether stereochemistry constraints get displayed on the screen.
show-vibrational-vectors: Variable, Read/Write.
Type: boolean.
Whether or not to display per-atom vibrational vectors.
solvate-system: Command.
Arg list: (void).
Solvate current system using default box size.
solvate-system-in-this-box: Command.
Arg list: float, float, float.
The three Float args give the size of the box for solvation.

source-refs-in-errors: Command.
    Arg list: (void).
    Controls presentation of filename, line number in error messages.
spheres-highlighted: Variable, Read/Write.
    Type: boolean.
    CPK overlapping spheres should be highlighted when shaded.
spheres-shaded: Variable, Read/Write.
    Type: boolean.
    CPK overlapping spheres should be shaded.
start-logging: Command.
    Arg list: string, boolean.
    Begin append of logging computation results to file String-1.
status-message: Variable, Read/Write.
    Type: string.
    The text of the last status message.
sticks-width: Variable, Read/Write.
    Type: integer in range (0 .. 25).
    Sticks rendering width in pixels.
stop-logging: Command.
    Arg list: (void).
    Don't log computation results any more.
stretch-energy: Variable, Readonly.
    Type: float in range (-1e+010 .. 1e+010).
    Results from backend computation.
switch-to-user-menu: Command.
    Arg list: (void).
    Change menu to the user customized menu.
toggle: Command.
    Arg list: string.
    Invert value of boolean variable.
torsion-energy: Variable, Readonly.
    Type: float in range (-1e+010 .. 1e+010).
    Results from backend computation.
total-energy: Variable, Readonly.
    Type: float in range (-1e+010 .. 1e+010).
    Results from backend computation.
translate-merged-systems: Variable, Read/Write.
    Type: boolean.
    Should newly merged/pasted molecules be translated off to one side?
translate-selection: Command.
    Arg list: float, float, float.
    Translate the selection by (dx, dy, dz)
translate-view: Command.
    Arg list: float, float, float.
    Translate the view by (dx, dy, dz).
translate-whole-molecules: Variable, Read/Write.
    Type: boolean.
    Select translation of entire molecule.
uhf: Variable, Read/Write.
    Type: boolean.
    Perform UHF(true) or RHF(false) calculation.
un-select-atom: Command.
    Arg list: integer, integer.
    Un-selects atom int-1 in molecule int2. Honors selection-target.
un-select-residue: Command.
    Arg list: integer, integer.
    Un-selects residue int-1 in molecule int-2, disregarding selection-target.

unconstrain-bond-angle: Command.
Arg list: (void).
Remove any angle constraint for the three currently selected atoms.
unconstrain-bond-length: Command.
Arg list: (void).
Remove any constraints on two currently selected atoms.
unconstrain-bond-torsion: Command.
Arg list: (void).
Remove any torsion constraint for the four currently selected atoms.
use-fast-translation: Variable, Read/Write.
Type: boolean.
Use bitmap for XY translations.
use-no-restraints: Command.
Arg list: (void).
Ignore all restraints.
use-parameter-set: Command.
Arg list: string.
Use parameter set string1 for current molecular mechanics methods.
use-restraint: Command.
Arg list: string, boolean.
(selection-name, if-use)
uv-band-count: Variable, Read/Write.
Type: integer.
The total number of uv bands.
uv-dipole-components: Variable, Read/Write.
Type: vector of float-list.
(i) The components of the dipole moment for the i-th state.
uv-energy: Variable, Read/Write.
Type: vector of float.
(i) The energy of the i-th uv band.
uv-oscillator-strength: Variable, Read/Write.
Type: vector of float.
(i) For the current state.
uv-spin: Variable, Read/Write.
Type: vector of float.
(i) The total spin of the i-th state.
uv-total-dipole: Variable, Read/Write.
Type: vector of float.
(i) The total dipole of the i-th excited state.
uv-transition-dipole: Variable, Read/Write.
Type: vector of float, float, float.
(i) The components of the transition dipole for the i-th state.
variable-changed: Command.
Arg list: string.
Declare that the named variable has changed.
velocities: Variable, Read/Write.
Type: array of float, float, float.
(iat, imol) The x, y, and z velocity components of atom iat in molecule imol.
velocities-in-hin-file: Variable, Read/Write.
Type: boolean.
Should velocities be written into a hin file?
version: Variable, Readonly.
Type: string.
The version number of HyperChem, read-only.
vibrational-mode: Variable, Read/Write.
Type: integer.
The index of the current normal mode.

view-in-hin-file: Variable, Read/Write.
Type: boolean.
Should view be written into a hin file? (useful for comparing hin files)
wall-eyed-stereo: Variable, Read/Write.
Type: boolean.
Wall eyed stereo.
warning: Variable, Read/Write.
Type: string.
The current warning.
warning-type: Variable, Read/Write.
Type: enum(none, log, message).
Destination for warning messages.
warnings-are-not-omsgs: Command.
Arg list: (void).
Specifies that warning messages are to appear in message boxes.
warnings-are-omsgs: Command.
Arg list: (void).
Specifies that warning messages should be treated like o-msgs.
window-color: Variable, Read/Write.
Type: enum(Monochrome, Black, Blue, Green, Cyan, Red, Violet, Yellow, White).
The background color for the window.
window-height: Variable, Read/Write.
Type: integer.
Height of the HyperChem window in pixels.
window-width: Variable, Read/Write.
Type: integer.
Width of the HyperChem window in pixels.
write-atom-map: Command.
Arg list: string.
Writes a mapping of backend atom numbers to HyperChem (atom, molecule) pairs.
write-file: Command.
Arg list: string.
String-1 names the file into which the current system should be written.
x-y-rotation-cursor: Variable, Read/Write.
Type: float angle in range (0 .. 3600).
Select X-Y axis rotation cursor tool.
x-y-rotation-icon-step: Variable, Read/Write.
Type: float angle in range (0 .. 3600).
Select X-Y axis rotation step.
x-y-translation-icon-step: Variable, Read/Write.
Type: float in range (0 .. 1000).
Select X-Y translation step.
z-rotation-cursor: Variable, Read/Write.
Type: float angle in range (0 .. 3600).
Select Z axis rotation cursor tool.
z-rotation-icon-step: Variable, Read/Write.
Type: float angle in range (0 .. 3600).
Select Z axis rotation step.
z-translation-cursor: Variable, Read/Write.
Type: float in range (0 .. 1000).
Select Z axis translation cursor tool.
z-translation-icon-step: Variable, Read/Write.
Type: float in range (0 .. 1000).
Select Z translation step.
zindo-1-pi-pi: Variable, Read/Write.
Type: float in range (0 .. 2).

Overlap weighting factors.
zindo-1-sigma-sigma: Variable, Read/Write.
Type: float in range (0 .. 2).
Overlap weighting factors.
zindo-s-pi-pi: Variable, Read/Write.
Type: float in range (0 .. 2).
Overlap weighting factors.
zindo-s-sigma-sigma: Variable, Read/Write.
Type: float in range (0 .. 2).
Overlap weighting factors.
zoom: Command.
Arg list: float in range (0.01 .. 50).
Set the magnification.
zoom-cursor: Variable, Read/Write.
Type: float in range (1 .. 1000).
Select zoom cursor tool.
zoom-icon-step: Variable, Read/Write.
Type: float in range (1 .. 1000).
Select zoom step.

Chapter 7
Type 2 (Tcl/Tk) Scripts

Introduction

This chapter describes a new scripting language for HyperChem that appears for the first time in Release 5.0. This language is called the *Tool Command Language* or just Tcl ("tickle") for short. Tcl was developed by Professor John Ousterhout and his students at the University of California, Berkeley and placed in the public domain. Hypercube has build the language directly into the HyperChem product. Tcl is a general purpose, but relatively simple, interpreted scripting language. It is *embeddable*. That is, one can extend the core Tcl language with additional commands; Hypercube has imbedded the full *HyperChem Command Language* (Hcl) into Tcl. This makes it an extremely powerful language for molecular modeling. A HyperChem script can now be made to do almost anything you wish with a relative modest amount of programming.

Since Tcl is imbeddable, it commonly comes with an extra set of commands that allow it to be used for building graphical user interfaces (GUI's). The extra module is called the *Toolkit* (Tk) and the combination is called Tcl/Tk. Within HyperChem it might be called Tcl/Tk/Hcl but we simply refer to it as Tcl/Tk (or just Tcl when we are not concerned with graphical elements).

Because the Tcl/Tk/Hcl combination is so powerful it is now possible to write whole graphical applications as simple scripts. They can be written quickly and they can be debugged quickly since Tcl is an interpreted language. One might first think that a scripting language is just a way of customizing HyperChem but there are many situations where Tcl can be used to write completely new applications, reusing whatever functionality HyperChem conveniently can provide. In other situations Tcl is a convenient glue language for interfacing HyperChem to your own applications, possibly written in C, C++, or Fortran. HyperChem contains many capabilities that you do not need or want to replicate, but can just use - via this new scripting capability.

Elements of Tcl

This manual cannot be a tutorial or reference for Tcl! The language is extensive and competent Tcl programmers must obtain the appropriate reference books and materials as with any other programming language. Nevertheless, with a few ideas described here and the examples that are provided, you should be able to write your first Tcl/Tk program and can begin to extend HyperChem, in some useful way, for your own purposes. Appendix B contains a very brief outline of the Tcl commands.

The obvious way to start learning the language is through books on the subject or through user groups and other material on the Internet and World Wide Web.

Books

The obvious book to use is the one by Ousterhout himself. It is the oldest, however, and may not be up to date compared with the latest releases of the software.

- John K. Ousterhout, *Tcl and the Tk Toolkit*, 1994, Addison-Wesley, Reading, Mass., ISBN 0-201-6337-X.
- Brent Welch, *Practical Programming in Tcl and Tk, Prentice* Hall, 1994.
- Eric F. Johnson, *Graphical Applications with Tcl & Tk*, 1996, M&T Books, New York, N.Y., ISBN 1-55851-471-6.

Internet

Tcl resources are also readily available on the internet. The Usenet newsgroup comp.lang.tcl may be useful to you. Some of the WWW sites where Tcl resources are available are:

- http://www.sunlabs.com:80/research/tcl/
- http://www.sco.com/Technology/tcl/Tcl.html
- http://www.elf.org/tcltk-man-html/contents.html

What is Tcl?

The Tcl language is an interpreted scripting language that, after you are familiar with it, is really quite simple and easy to use. However, to chemists and those with limited programming experience outside C and Fortran, it may seem somewhat strange to begin with. The language may seem to you to be very oriented toward strings rather than numbers, but this is perhaps one of its strengths.

Commands and Arguments

Tcl scripts (`*.TCL`) consist of a sequence of Tcl commands. Each Tcl command consists of the name of the command followed by its arguments:

```
command <argument1> <argument2> ...
```

Every command returns a *string*. For example, the command,

```
incr $x 3
```

increments the *value* of x by three and returns the string corresponding to the new value..

Variables and Values

Tcl allows you to store values in *variables* and use values in commands. For example, the command,

```
set x 3.5
```

stores the value 3.5 in the variable x. The value of x is the string, “3.5”, obtained by using the syntax, $x. To perform arithmetic like operations, one must use a command, *expr*, that concatenates its arguments into a single string, evaluates it as an arithmetic expression, and converts the numerical result back into a string to return it. Thus,

```
expr $x * 3
```

returns the string, “10.5”.

Command Substitution

This allows you to use the result of one command (a string) as an argument in another command. Thus,

set y [expr $x + 0.5]

is the way to set y to the value of 4.0, or in general the way to perform arithmetic. The square brackets invoke command substitution, i.e. everything

inside the square brackets is evaluated as a separate Tcl script and the result of the script is substituted in place of the bracketed command.

Procedures and Control Structures

It is possible to write Procedures in Tcl. For example, a procedure called power which when passed x and n computes x^n is as follows:

```
proc power {x n} {
    set result 1
    while {$n > 0} {
        set result [expr $result * x]
        set n [expr $n - 1]
    }
    return $result
}
```

The braces, { }, are like the double quotes that you place around words that have spaces in them to make them into single words - “two words”, for example. Braces nest, i.e. the last argument of the proc command starts after the open brace on the first line and contains everything up to the close brace on the last line. No substitutions occur within braces - all the characters between the two braces are passed verbatim to proc as its third argument.

Tk

The Tk commands that are added to Tcl have to do with creating and using a set of widgets that appear in a window that appears whenever you invoke a Tcl script (unless you place the Tcl command, *TclOnly*, at the start of the script). The widgets include Labels, Text Boxes, Buttons, Menus, Frames, Scales, Radio Buttons, Check Boxes, List Boxes, etc. A widget is created by a command as follows:

```
<widget-type> <widget-name> <argument1> ...
```

The widget name traditionally starts with a “dot”. For example, a button is created as follows:

```
button .b -text “Push Me” -command { <action> }
```

This button with the text “Push Me” on it will appear in the GUI and will cause the Tcl command <action> to be executed when it is pushed.

A necessary Tk command in a Tcl script is the one which lays out the widgets. This pack command,

```
pack .b
```

lays out each of its widget arguments in the window, in the order in which the arguments are named. By default a vertical ordering is used.

Hcl Embedding

Finally, Type 2 or Tcl/Tk scripts can all execute any of the HyperChem script commands in the HyperChem Script language (Hcl). Within a Tcl script these Hcl commands are divided into two types:

- Executable command - hcExec
- HSV Query - hcQuery

hcExec

The Tcl command for executing a Hcl command is hcExec. For example,

hcExec “do-molecular-dynamics”

hcExec “window-color green”

hcExec “menu-file-open”

hcExec “query-value window-color”

This command can be used to execute any Hcl script, including menu activations, direct commands, and any HSV read or write. However, an HSV read such as, “query-value window-color”, will place the return message into a HyperChem dialog on the screen as if it was a normal Hcl script and not return anything to the Tcl script. The string that is returned from any hcExec command is always the empty string, “”. If one wants a Tcl script to manipulate and use HSV results from HyperChem then the hcQuery command should be used.

hcQuery

The Tcl command for querying HyperChem for the value of an HSV is hcQuery. For example,

```
set x [hcQuery window-color]
```

or

```
set x [hcQuery “window-color”]
```

The quotes are optional in this context. The hcQuery command is a normal Tcl call and returns a string as it should. Thus, if one queries an integer value, a string is returned that represents the integer; everything is consistent. For example, if the number of atoms in molecule 3 is requested,

```
set number_atoms [hcQuery “atom-count 3”]
```

then this value can be later used as in the following:

```
set twice_number_atoms [expr $number_atoms * 2]
```

Examples

While we have already seen a couple of very simple Tcl scripts in earlier chapters, this section describes three examples to help you get started in programming Tcl/Tk. The best way to learn the language is to use it. After understanding these examples, you should try a few scripts yourself. Obviously, if you are to become relatively expert in this subject you will need to obtain some books on Tcl or get assistance through the Internet in understanding the full syntax and semantics of the language.

Calculating the Number of Atoms

You have seen in earlier chapters some simple TclOnly scripts. This script is our first example of a Tcl/Tk script that uses Tk to generate a GUI. The example is very simple and is used to show the number of atoms in the HyperChem workspace. The code is very short,

```
entry .en -textvariable natoms -width 20
button .b -text "Calculate Atoms" -width 20 -command {
set natoms [hcQuery "atom-count"]
}
pack .b .en
```

This script has a GUI with two widgets. The first is a text entry widget, referred to in Tcl/Tk as an *entry*. It can be used to enter text or, as in this case, display text. We are going to use it to display the number of atoms in the HyperChem workspace. The text in an entry widget can be bound to a Tcl variable such that the text that appears always represent the current value of

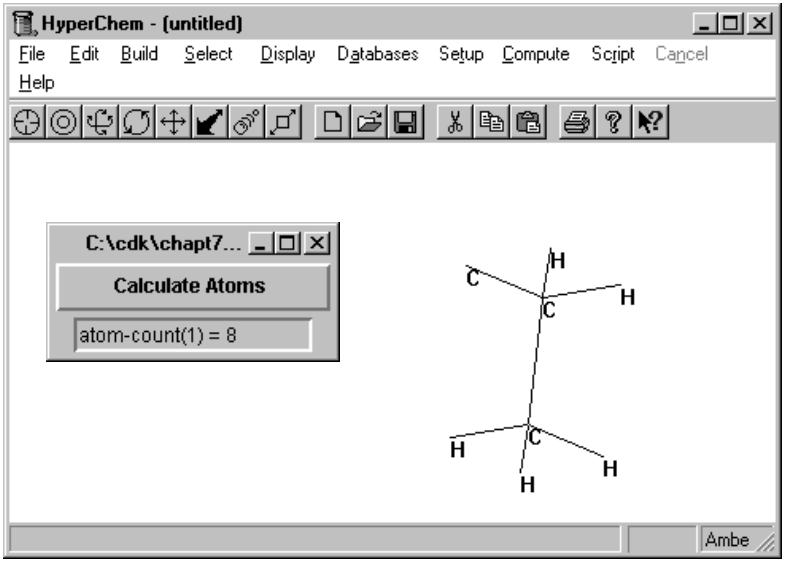
the variable. This is what we have done here; we have bound the variable *natoms* to the entry widget.

The next widget is just a button widget as we have discussed above. When the button is pushed, it executes the Tcl command,

```
set natoms [hcQuery atom-count]
```

which assigns a value to natoms that is the result of the query to HyperChem for the value of the HSV, *atom-count* (the elimination of quotation marks around *atom-count* is deliberate to show that they are optional). Now, *atom-count* is a vector and since we have left off the index describing the molecule number, i.e. the specific component of the vector, all vector components will be returned. If there are two molecules in the workspace, we will get two components returned, etc. Because natoms is bound to the entry widget, the value returned from pushing the button will appear in the entry widget.

Finally, the two widgets must be packed, i.e. they must be laid out on the window. The packing mentions the button first and then the entry so that this is the order in which they appear in the window. The default layout scheme is to have them arranged vertically. The result of pushing the button is thus:



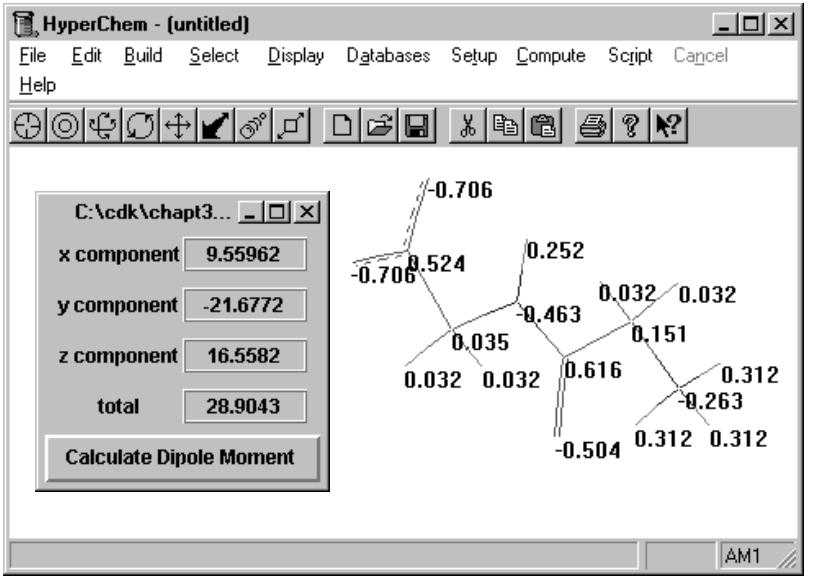
The title at the top of the Tcl/Tk window is the title of the script that has been executed. Only one molecule is present so only the first component of the vector is returned. If the Hcl script command, *query-response-has-tag false*,

had been executed in the Tcl script prior to the query for atom-count, then the Tcl/Tk entry in the window would only show “8” as the tag, *atom-count(1)*=, would have been eliminated.

To elaborate on this kind of script, you simply need to add more widgets and perform more complicated actions when GUI elements like buttons are pressed.

Calculating a Dipole Moment

This example Tcl/Tk script can be used to calculate a dipole moment for any situation where there are atomic charges. The script is available on the HyperChem CD-ROM as `dipole.tcl`. The script makes the assumption that the dipole moment can be calculated from the partial atomic charges only. These are the charges you see when you ask for charge labels in HyperChem. This script would perhaps be useful for molecular mechanics where HyperChem currently does not calculate a dipole moment. If one uses the following Tcl script to calculate a dipole moment from the Amber template charges of the glycine dipeptide zwitterion, one gets,



The Tcl script for this calculation will now be described. The first part of the script is associated with setting up the eight label widgets for the text describing the four values and for the four values themselves.

Labels

We are going to use labels to describe everything in this window apart from the button. The labels need to be laid out both horizontally and vertically. The easiest way to do this is to place labels horizontally into a frame and then arrange the frames vertically. We use a simple flat label for the text describing which dipole quantity we are talking about and a label with a relief ridge around it for the showing the value. Thus, the first frame is just:

```
frame .f1
label .f1.l1 -text "x component" -width 20
label .f1.l2 -textvariable xdipole -width 20 -relief ridge
pack .f1.l1 .f1.l2 -side left
```

The label .l1, for example, is a component of the frame .f1 as indicated by the notation .f1.l1. The packing is done horizontally starting at the left of the window. This first frame is for the variable that we have called xdipole, the x-component of the dipole moment. The other three frames are identical:

```
frame .f2
label .f2.l1 -text "y component" -width 20
label .f2.l2 -textvariable ydipole -width 20 -relief ridge
pack .f2.l1 .f2.l2 -side left

frame .f3
label .f3.l1 -text "z component" -width 20
label .f3.l2 -textvariable zdipole -width 20 -relief ridge
pack .f3.l1 .f3.l2 -side left

frame .f4
label .f4.l1 -text "total" -width 20
label .f4.l2 -textvariable totdipole -width 20 -relief
ridge
pack .f4.l1 .f4.l2 -side left
```

Finally, we pack the four frames from top to bottom. We will use a slightly modified call to pack here, called the “configure” option which allows more formatting which, in this case, we use to add a little more padding between the frames.

```
pack configure .f1 .f2 .f3 .f4 -pady 10
```

Button

The next part of the code describes the button and the action taken when the button is pressed. The button is called .b and the action is everything between the opening brace on the first line and the closing brace on the last line. The first thing that is done is to execute a Hcl command to see that the value we want comes back without a tag that would interfere with its direct use in Tcl commands. We then initialize the component values of the dipole moment we are going to calculate, and initialize the loop counter i. We then get the count of the number of atoms in molecule 1 for the loop over atoms. The script we are using here is limited to calculating only the dipole moment of molecule 1. It is an exercise for the reader to extend this to calculate the dipole moment of the whole molecular system when it has more than one molecule in it. The loop is then entered.

```
button .b -text "Calculate Dipole Moment" -command {
hcExec "query-response-has-tag false"
set xdipole 0
set ydipole 0
set zdipole 0
set i 0
set natoms [hcQuery "atom-count 1"]
while { $i <$natoms } {
incr i 1
```

The principal task inside the loop is to get the coordinates and charge of each atom and to multiple them together, accumulating them as we go through the loop. The charge is straight-forward as the first line below indicates. However, the coordinates of an atom come back as a string representing the three x,y,z components separated by commas. This string, referred to here as sz, must be parsed to extract the individual components which are placed in arrays x, y, and z. The parsing is accomplished with string commands that extract the length of a string (string length), look for particular substrings such as “,” starting at the beginning of the string (string first), extract a substring

out of a string (string range), search a string from the end to the beginning looking for a substring (string last), or trim leading blanks off of a string (string trimleft).

```
set charge($i) [hcQuery "atom-charge($i,1)"]
set sz [hcQuery "coordinates($i,1)"]
set sz_length [string length $sz]
set comma1 [string first "," $sz]
set x($i) [string range $sz 0 [expr $comma1 - 1] ]
set x($i) [string trimleft $x($i)]
set comma2 [string last "," $sz]
set y($i) [string range $sz  [expr $comma1 + 1] [expr $comma2 - 1] ]
set y($i) [string trimleft $y($i)]
set z($i) [string range $sz [expr $comma2 + 1] $sz_length ]
set z($i) [string trimleft $z($i)]
```

This parsing may seem a bit difficult but this is about the worst that it ever gets. Finally, the charges and coordinates are multiplied and accumulated.

```
set xdipole [expr $xdipole + $charge($i) * $x($i)]
set ydipole [expr $ydipole + $charge($i) * $y($i)]
set zdipole [expr $zdipole + $charge($i) * $z($i)]
}
```

Once outside the loop, the dipole moment components are converted to Debyes and then the total is computed from the components.

```
set factor 4.8033
set xdipole [expr $xdipole * $factor]
set ydipole [expr $ydipole * $factor]
set zdipole [expr $zdipole * $factor]
```

```
set totdipole [expr $xdipole * $xdipole]
set totdipole [expr $totdipole + $ydipole * $ydipole]
set totdipole [expr $totdipole + $zdipole * $zdipole]
set totdipole [expr sqrt($totdipole)]
```

The last thing is to end the button code and to pack the button below the labels.

```
}
pack .b
```

This completes the calculation of the dipole moment. Additional Tcl scripts are available from the HyperChem CD-ROM.

Chapter 8
DDE Interface to HyperChem

Introduction

This chapter describes the basics of a *lower-level* Dynamic Data Exchange (DDE) interface to HyperChem and illustrates it with interactions between Microsoft Word or Microsoft Excel and HyperChem.

DDE versus HAPI

The DDE interface is referred to here as lower-level because Hypercube's new CDK has introduced a *higher-level* interface that sits on top of DDE and is referred to as the HyperChem Application Programming Interface (HAPI). The HAPI way to interface programs to HyperChem is fully described in Chapter 11. One point about making HAPI calls rather than DDE calls is that it has abstracted away from any machine or operating system dependency on DDE and can be used in a UNIX environment, etc. Any investment in HAPI is preserved since HAPI will be re-implemented by Hypercube if new lower levels of interprocess and interprocessor messaging replaces DDE.

Use of DDE in Windows Applications

The DDE interface is more strictly *operating system dependent* than it really is very low-level. We will illustrate it here with a word processor and a spreadsheet. The next chapter illustrates its use within Visual Basic programs. These DDE interfaces are actually relatively high-level interfaces and are available for multiple word processors and spreadsheets. The DDE interface is very standard for Windows programs and many manufacturers have adopted it. It is almost non-existent outside the Microsoft Windows environment, however. Nevertheless, almost any Windows program will have a DDE interface and HyperChem can have conversations and exchange data with a large number of other standard Windows programs. We illustrate the CDK technology using DDE interactions between HyperChem and Word, Excel or

Visual Basic, because these are very prominent programs of their class not because identical interfaces could not be demonstrated for other word processors, spreadsheets, etc.

There are indications that Microsoft might eventually replace DDE with something referred to as OLE Automation. Essentially identical considerations would really apply to this new approach compared with the discussions given below for DDE. Nevertheless, a newer version of HyperChem might some day be needed to replace HyperChem's current use of DDE with OLE Automation. There are no indications that Microsoft will abandon its use of DDE anytime in the near future.

Basic Properties of DDE

Dynamic Data Exchange is a mechanism for interprocess communication in Windows and NT. Two separate programs carry on a DDE conversation by sending messages to each other. The two programs are referred to as a client and a server. A DDE server generally has data or services that may be of interest to another program. A DDE client wishes to obtain such data or services. In our context, HyperChem is a DDE Server and makes its data and services available to other programs. A client such as Microsoft Word or Excel, or your program, initiates a communication with the HyperChem Server, exchanges data with it, and sends it messages to control its actions.

Two programs that engage in DDE conversations with each other need not be specifically coded to work with each other. A DDE Server, such as HyperChem, will publicly define its messaging protocol so that all clients know in advance how to obtain services from the server.

A protocol begins with a server defining the name of its *application*, a *topic* for the conversation, and an *item* that defines the exchange. In most cases the item is a piece of data to be read from the server.The client initiates the conversation by using the application and topic to establish a communication link In our case, the application is *HyperChem*, the topic is always *System*, and the item is the name of an HSV. In addition, commands can be sent to the server and this is the way menu activations, direct commands and HSV writes are performed.

DDE Message Types

Each DDE interaction or message passed between client and server is a member of a small number of types. The relevant message type for messages sent from your client program to the HyperChem server are as follows:

DDE_INITIATE

This message, broadcast by a client, requests a conversation with an application on a topic. Hyperchem will respond if the application is “HyperChem” and the topic is “System”.

DDE_ EXECUTE

This is the message sent by a client to HyperChem to have it execute a piece of text corresponding to a script command. Thus the content of such a message might be, “window-color green” or “do-molecular-dynamics” to cause the background HyperChem window to turn green or to initiate a molecular dynamics trajectory.

DDE_ REQUEST

This message type is the traditional client request for a piece of data. The item named in the request is the data being requested, such as “window-color”, “scf-binding-energy”, etc.

DDE_ ADVISE

This message type is sent when the client is requesting a hot link for an item. This means that the server will return an HSV automatically on the hot link whenever the data representing the HSV changes within HyperChem.

These ideas can be illustrate with simple Excel and Word macros or, as in the next chapter, by simple Visual Basic Programs.

DDE Interface to Microsoft Word

Microsoft Word and other Windows word processing programs can talk DDE to other Windows programs. These conversations can be used to make a document come to life with illustrations, etc. For example, a tutorial on some aspect of chemistry could be written in Microsoft Word. The manuscript could contain buttons which the reader could push as they were reading about a topic. These would cause a HyperChem window on the same screen to perform calculations on, or illustrations of, the molecular systems being described in the manuscript.

In this and following sections we will describe the topic of DDE communication with a very trivial illustration that just changes the color of the Hyper-

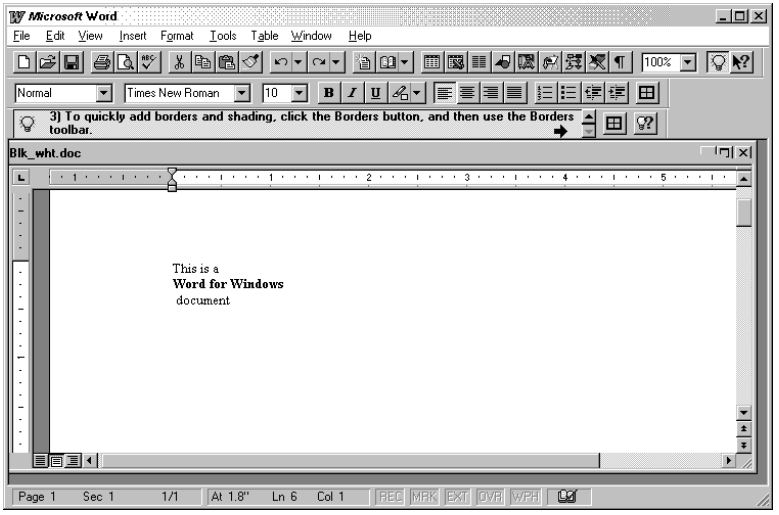
Chem screen. This example has very little chemical meaning but it is *visual* and easily seen when you try it. It is characteristic of hundreds of other available HSV's that do have chemical meaning and it is an example of an HSV that is both readable and writable. You should be able to extrapolate from this trivial example, that illustrates the basic idea, to examples that perform more meaningful chemistry in line with your own teaching or research interests.

Red and Green Example

The following screen shows a Microsoft Word document.

1. Bring up a copy of Microsoft Word and type a few lines of text into it.

.



What we are going to do is add two buttons to the text that you can click on to effect actions in HyperChem. Let us first of all create two Word Macros. Word comes with a dialect of Word Basic that you can use to customize Word and write macros. These macros are then associated with a particular template file (`*.dot`). When that template is in use, these macros are available.

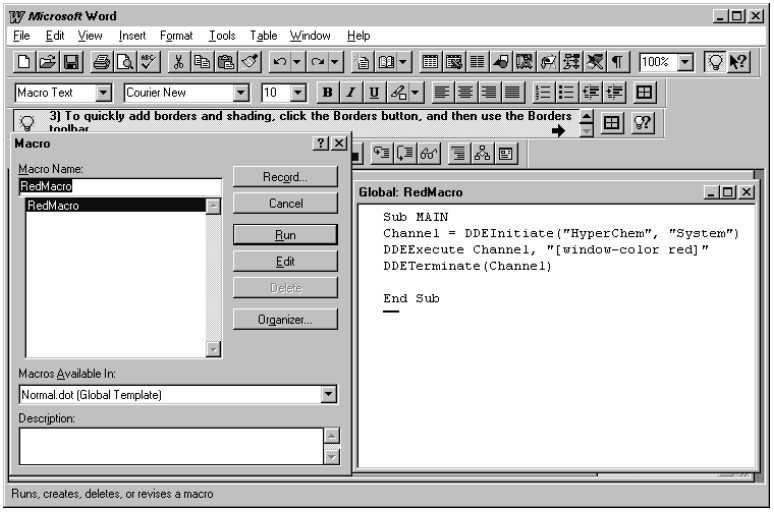
2. Select the <Tools/Macro...> menu item and create a macro with the name, RedMacro. That is, type RedMacro in the text box, "Macro Name", and choose a convenient template file to use for "Macros Available In:", and then push the "Create" button.

3. Add the following Word Basic code to create the macro,

```
Channel = DDEInitiate("HyperChem", "System")
```

```
DDEExecute Channel, "[window-color red]"
DDETerminate(Channel)
```

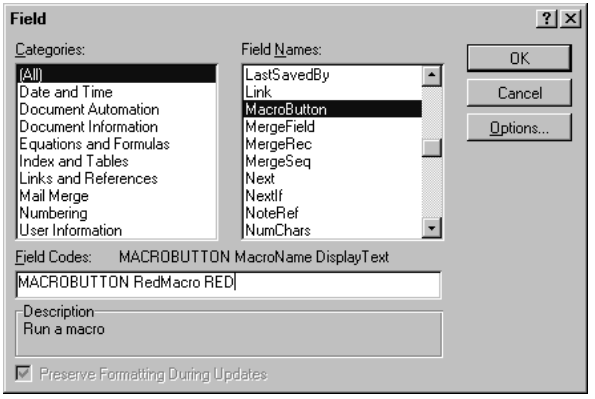
You should see a screen such as the following.



The macro code simply initiates communication with the application, HyperChem, on the topic of System and then sends a DDE_EXECUTE message to HyperChem containing the text that corresponds to a HyperChem script command, which here sets the background window color to red.

Next you need to implement something in the Word document that triggers the Macro. Assuming the text cursor in the document is where you want it,

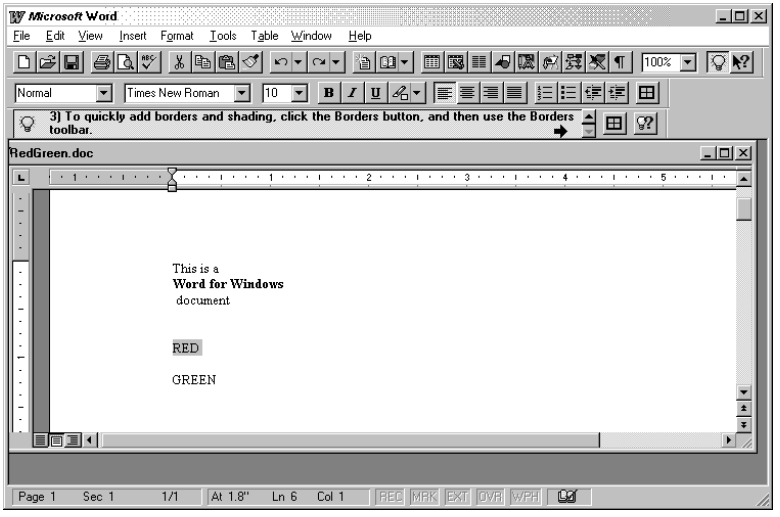
4. Select the menu item <Insert/Field...> to bring up the following dialog box.



5. Select the “Field Name” corresponding to *MacroButton* as shown and then type arguments *RedMacro* and some arbitrary text, such as *RED*, into the bottom text entry box, as shown, prior to hitting OK.

6. Repeat the whole process for a “Green button”.

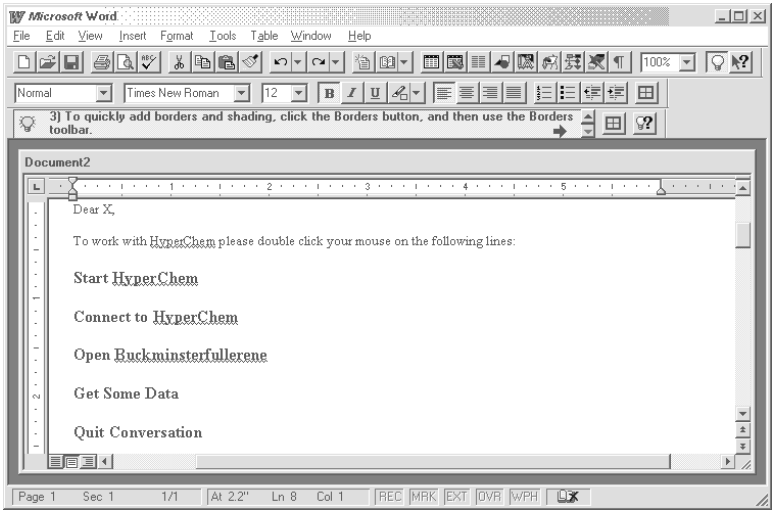
You should now see something resembling the following.



If you now bring up HyperChem and double click on either of the two “buttons”, i.e. the text RED or the text GREEN inside the Word document you will see these color changes in HyperChem.

Extended Example

The first example of Word Basic above was about as simple as one can get. Word Basic has more programming capability than that example showed. A slightly more elaborate example can show some of this programming capability. The following five macros that we are about to describe were generated exactly as above and correspond to the large bold-faced type of the document shown below:



The five macros used here are:

ActivateHC

```
Sub MAIN
If (AppIsRunning("HyperChem") = 0) Then
LineDown
Insert "Starting HyperChem ..." + Chr$(13)
Shell "c:\hyper\chem.exe", 0
LineUp
EndIf
End Sub
```

Each of these macros has a MAIN subroutine. You may use additional subroutines below, or above, the MAIN subroutine. This macro starts HyperChem on the assumption that it is in the directory `c:\hyper`. Your copy

may not be in this same location. The Insert command puts text at the current cursor and LineUp and LineDown move the cursor.

ConnectHC:

```
Sub MAIN
channel = DDEInitiate("HyperChem", "System")
If (channel <> 0) Then
c$ = Str$(channel)
SetDocumentVar "channel", c$
LineDown
Insert "Document <-> HyperChem connection is ready ..."
Else
LineDown
Insert "Cannot connect to HyperChem !!!"
EndIf
End Sub
```

The above macro starts HyperChem and stores the communication channel in a variable that can be accessed by other macros.

ExecuteCmd

```
Sub MAIN
c$ = GetDocumentVar$("channel")
channel = Val(c$)
name$="c:\hyper\c60.hin"
AppMaximize "HyperChem", 1
DDEExecute channel, "open-file " + name$
DDEExecute channel, "align-viewer z"
DDEExecute channel, "align-molecule primary, x, tertiary, z"
DDEExecute channel, "menu-display-scale-to-fit"
DDEExecute channel, "zoom 1.4"
DDEExecute channel, "menu-edit-copy-image"
LineDown
EditPaste
End Sub
```

The above macro is an example of one that opens a file containing Buckminsterfullerene (C_{60}), manipulates it in HyperChem and then copies and pastes it into the document.

GetData

```
Sub MAIN
     c$ = GetDocumentVar$("channel")
```

```
channel = Val(c$)
atom_count$ = DDERequest$(channel, "atom-count")
atomic_symbol$=DDERequest$(channel, "atomic-symbol")
LineDown
Insert atom_count$
Insert atomic_symbol$
End Sub
```

The above macro first retrieves the text representation of the HyperChem vector variable, *atom-count*, which represents the number of atoms in each molecule. The second query retrieves the atomic symbols for the atoms in the workspace (C for Carbon, in this case).

DisconnectHC

```
Sub MAIN
c$ = GetDocumentVar$("channel")
channel = Val(c$)
DDETerminate channel
End Sub
```

This last macro terminates the connection to HyperChem.

The macros above are stored as part of a Microsoft Word TEMPLATE, not as a part of the document. So, as long as you are using the same template, you do not need to worry about macros: they are always in place. However, if you move your document to another machine, or if you want to distribute it, you must remember to move the template also. You may save a template in a DOT file of Microsoft Word.

Install the macros and try them.

DDE Interface to Microsoft Excel

Microsoft Excel is an example of a spreadsheet that has extensive capabilities for DDE conversations with HyperChem. You can, for example, set up tables of molecules and automate the computation of molecular properties so that they show up in tables within the spreadsheet. The capability for using a spreadsheet in conjunction with HyperChem will only be very briefly be illustrated here, as another example of the richness of the CDK and the open architecture of HyperChem. This capability has been there since HyperChem's inception and has been fairly commonly exploited so it will not be dwelled on here.

With Release 5 and the CDK, the spreadsheet capability has been enhanced because of the addition of enhanced scripting, i.e. now Tcl/Tk scripts in addi-

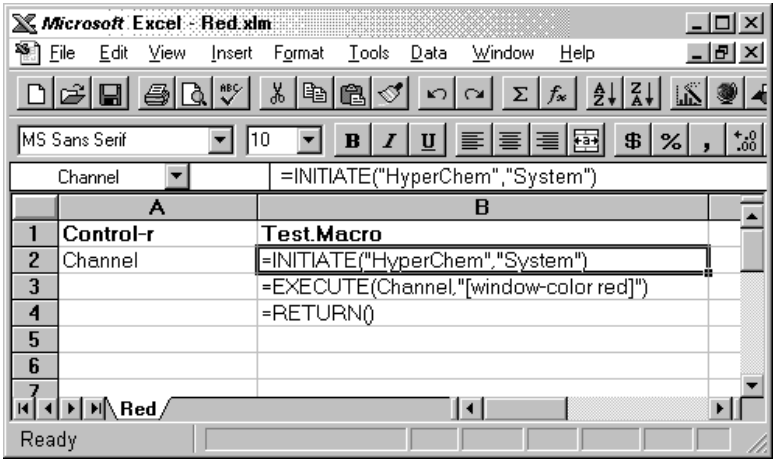
tion to Hcl scripts as with earlier versions of HyperChem. However, Excel macros have considerable programming and GUI capability so that if one prefers to program within a product like Excel itself, it might be that the enhanced functionality of HyperChem along with more Hcl scripts is more important to you than the addition of Tcl/Tk.

We emphasize Microsoft Excel here as our spreadsheet example but others spreadsheets such as Lotus 123, Quatro Pro, etc. also have DDE capability and could be used to communicate with and exchange data with HyperChem. The examples here are restricted to Excel, however.

We will illustrate the interface of HyperChem and Microsoft Excel using the same example as we began with in describing Microsoft Word, i.e. the trivial example that sets an HSV (window-color).

Red (and Green)

A very simple Excel Macro is the following:



Running this macro, `RED.XLM`, will change the color of the HyperChem window to red. It can be run by simple hitting Ctrl-r on the keyboard (provided Excel is running) or from the <Tools/Macro...> menu item. As in earlier examples, we simply initiate a conversation with the *application*, HyperChem, on the *topic*, System. We then execute a DDE command that is the Hcl script command, *window-color red.* A similar macro could change the color to green but with a different keyboard accelerator such as Ctrl-g. Then, alter-

nately hitting Ctrl-r and Ctrl-g would flash the HyperChem window from red to green to red..., etc.

The macro DDE commands, EXECUTE and REQUEST are fundamentally all you need to communicate with HyperChem. Other macro programming is needed, of course, to deal with the data coming from or going to HyperChem.

Additional Macros

The Excel macro language and Excel itself provide a powerful capability in combination with HyperChem but will not be described in detail here. The HyperChem Getting Started and Reference manual provide additional detail associated with the interaction between HyperChem and Excel and examples *.XLM files have been distributed with HyperChem since its inception. The ChemPlus product includes the code for an extensive Excel macro that makes 3D Ramachandran-like plots of the energy of a molecule versus two independent structural variables. You should also check the Hypercube WWW site (http://www.hyper.com) where Excel macros, along with scripts, are made available to users of HyperChem.

Chapter 9
DDE and Visual Basic

Introduction

This chapter describes the DDE interface between HyperChem and Microsoft's Visual Basic (VB). VB is chosen because it provides a very fast way of building extensions to HyperChem. While VB is not the only tool in its class, it is certainly one of the better ones. Some of the prejudice against Basic as a serious programming language remains, among scientific programmers and others. VB, however, is a serious modern tool that can allow you to very quickly put together applications, particularly applications involving graphical user interfaces, in a few hours - applications that used to take weeks or longer. For the demonstrations of the chapter, we will use Visual Basic 4.0 although earlier releases are also appropriate to use.

VB for GUIs or Computation

While one can certainly write whole applications in Visual Basic, its object-oriented character and uniqueness are illustrated best when quickly putting together a graphical user interface(GUI) rather than scientific number crunching types of code. A potential compromise is to build dynamic link libraries (DLL's) with C, C++, or Fortran and just have the GUI built in Visual Basic. A VB program can call a DLL, and you can choose the best of both worlds by rapidly protyping a user interface in VB and using legacy code in Fortran, for example.

VB with DDE or HAPI Calls

Since VB can call a DLL, it can call the DLL that defines the HyperChem Application Programming Interface (HAPI.DLL). The subject of the HAPI library, its calls, and how to use this method of interfacing to HyperChem is the subject of Chapter 11 and the subsequent chapters which give examples

of the HAPI library being used in various contexts. Chapter 11 includes a brief example of using HAPI calls with VB.

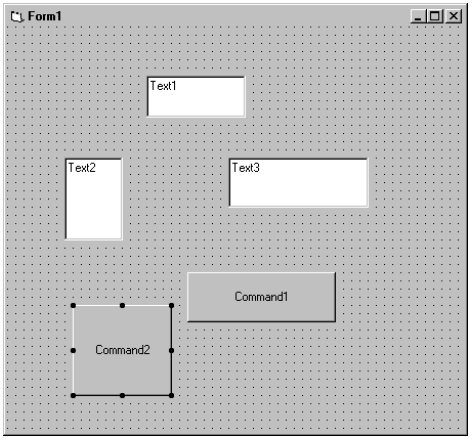
This chapter will focus on the low-level DDE interface to HyperChem for VB programs but we will also discuss very briefly the alternative HAPI way to build an interface between HyperChem and VB.

Red and Green

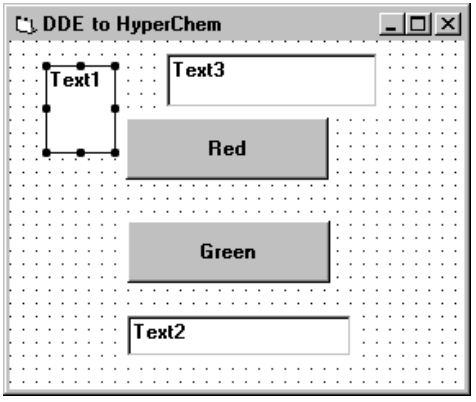
Our first example of a Visual Basic program communicating with HyperChem is, by now, our familiar and trivial example of a program that changes the background color of a HyperChem window. This particular example, however, also monitors the current color and displays the color, whether it is changed by the VB program or through the <Preferences> dialog box in HyperChem.

Basic Form and Controls

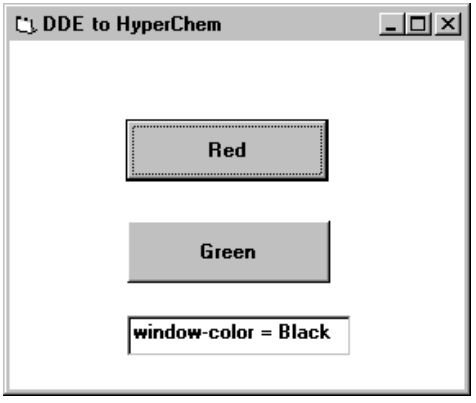
The VB project file for this example is called `REDGREEN.VBP` and is on the HyperChem CD-ROM. It was created by just opening Visual Basic and adding to the main form, three text boxes, Text1, Text2, and Text3 plus two command buttons, Command1 and Command2.



It is possible to lay out these linkcontrols and start from scratch or you can just read `REDGREEN.VBP` from the HyperChem installation directories,



What we have done here is to change the caption of Form1, the main form, to "DDE to HyperChem" and the caption of the two command buttons to "Red" and "Green". We have also positioned the controls that have the Visible property set to "true", i.e. Text2, Command1, and Command2. We have also changed the background color of Form1 to white and shrunk it a bit. A running version of the program might look as follows:



Start Up (Load)

The most important code in this example is that which on start up establishes the basic link with HyperChem and retrieves the initial color of the window (which will be restored on exit). The following is the code that is executed when Form1 loads,

```
Private Sub Form_Load()
If Text1.LinkMode = 0 Then
    Text1.LinkTopic = "HyperChem|System"
    Text1.LinkMode = 2
End If

    Text1.LinkItem = "window-color"
    Text1.LinkRequest

    Text2.Text = Text1.Text

If Text3.LinkMode = 0 Then
    Text3.LinkTopic = "HyperChem|System"
    Text3.LinkItem = "window-color"
    Text3.LinkMode = 1
End If

End Sub
```

With Visual Basic, a communication channel to HyperChem becomes associated with a control such as one of the text boxes on the form, i.e. Text1 or Text3. Once such a channel is set up, text being passed back and forth between the VB application and HyperChem is stored in VB, as the text field associated with the control, i.e. as Text1.Text or Text3.Text. These text fields are where Hcl script commands are placed prior to their execution and where any HSV messages coming from HyperChem end up.

A Cold Link Request

The LinkMode field for a text object describes the type of channel that is established or exists with another application (DDE server) like HyperChem. A value of 0 indicates no link or channel. Setting the value to 2 establishes a cold link for the channel such that values will be returned from HyperChem only when requested. The first operation on start up is thus to use the Text1 object to establish a cold link with HyperChem on the generic topic, “System”. Subsequently, a message request is made of HyperChem for the HSV, window-color. The return message from HyperChem containing the value of the HSV, *window-color*, is automatically placed into the text field of the Text1 object, i.e. as Text1.Text. The Text1 object has the visible property set to false so that it is not visible on the running application. Another text box, Text2, is used to display the value of the current window color. The appropriate operation above is to just pass Text1.Text into Text2.Text so that it is shown to the user.

A Hot Link

The Text3 object in this example is used to establish a “hot link” to HyperChem, associated with LinkMode = 1. This means that whenever HyperChem detects that *window-color* changes its value, from whatever source, HyperChem will notify whoever is listening of the new value. Setting LinkMode equal to 1 establishes this hot link to HyperChem requesting a desire to listen to the LinkItem.

The value returned to the VB application by HyperChem goes into the text field of Text3 and this field is automatically updated whenever the value changes in HyperChem. This placing of the new value into Text3.Text can be detected using the routine, Text3_Change(). The code for this routine in our example is,

```
Private Sub Text3_Change()
Text2.Text = Text3.Text
End Sub
```

This just takes the current value of the color and places it into the Text2 object so that it is visible to the user of the VB application as part of the VB window (form).

Execute

Next, we look at the code behind the two buttons that change the color of the HyperChem window to Red or Green when they are “clicked”. The code for the red command button is,

```
Private Sub Command1_Click()
Text1.LinkExecute ("window-color red")
Text2.Text = "window-color = Red"
End Sub
```

This code, simply sends a Hcl script command message to HyperChem. Here it changes the window color to red but it could be any script command from the HyperChem Command Language (Hcl) except that it should not be a query of an HSV (Use LinkRequest for these). In addition, the color displayed locally in Text2 is updated.

Unload

The final portion of this example is the code that is executed on exit from the VB application. This is,

```
Private Sub Form_Unload(Cancel As Integer)
```

```
Text1.LinkExecute (Text1.Text)
End Sub
```

This code return the color of the HyperChem window to its original color prior to executing the VB application. This color is still stored in Text1.Text as per the original request on loading of the form.

A HAPI Interface to VB

The Visual Basic example above showed how to make DDE calls to HyperChem, via an object such as a text box that provides the DDE channel or link. It is very object oriented but not an obvious way to do things for conventional C and Fortran programmers. Nor is it immediately obvious how to transfer binary data in this fashion.

The HAPI interface for VB consists of a set of straight-forward calls (either for text or binary data) that can be implemented in ordinary Basic code. The calls, such as *hbExecBin* which executes a binary form of a Hcl command, all begin with “hb”. Thus hbExecTxt is the text equivalent of the above. These and all the other HAPI calls that refer to Hcl script commands take as their first argument a long integer, defined in the file `HSV.BAS`. This integer maps to one of the many Hcl menu activations, direct commands, or HSV read/writes. For these calls, the name of the long integer is the normal Hcl name, e.g. *window-color*, but with hyphens replaced by underscores. That is, the relevant VB name to use for HAPI calls is *window_color*. Thus,

```
Dim Value As Double
Dim Result as Long
Value = 3.5
Result = hbSetReal( dipole_moment, value, 8)
```

is the code which would constitute a binary write of the value 3.5 to the HyperChem dipole moment. The integer variable “dipole_moment” is defined in `HSV.BAS`. This call is really targetted at a corresponding C language routine, hcSetReal, contained in the dynamic link library, `HAPI.DLL`. The file `HAPI.BAS` makes the appropriate declaration to tie hbSetReal to this DLL and should be included in all your Visual Basic applications that wish to use the HyperChem API.

Thus, any of the HAPI calls of Chapter 11 or Appendix C can be used in Visual Basic programs. All that is necessary is to include two files, `HSV.BAS` and `HAPI.BAS`, in your project and to make calls to routines labelled hb... rather than to the hc... calls of the C/C++ language or to the hf... calls of For-

tran. Appendix C gives the details of each HAPI call including how to declare and use them from Visual Basic.

Chapter 10
External Tcl/Tk Interface

Introduction

Hypercube has built a Tcl/Tk interpreter directly into HyperChem Release 5. This interface derives from Version 7.5 of Tcl and Version 4.1 of Tk. This *internal* interpreter is probably the most convenient way to use Tcl/Tk in conjunction with HyperChem, i.e. by either opening a `*.tcl` file or by executing a Hcl script command, *read-tcl-script*. There is, however, an alternative way to use Tcl/Tk that is called the *external* interface.

The external use of Tcl/Tk implies that you use a Tcl/Tk program or interpreter that is completely separate from HyperChem and may even be of a different version than the one used in HyperChem. This external Tcl/Tk interpreter can then be augmented in a standard way to add the HyperChem Command Language (Hcl) as an embedded extension. The external Tcl/Tk program communicates ultimately with HyperChem via Dynamic Data Exchange (DDE). But, as a user, you need only to make "HAPI calls" just as with the internal interpreter. That is, hcExec and hcQuery are still how you invoke Hcl commands with external Tcl/Tk.

Why External?

The interpreter for Tcl/Tk which is built into HyperChem offers an extremely powerful extension to the HyperChem Command Language. However there are certain situations where you might prefer external Tcl/Tk access to HyperChem. Such situations arise when:

1. You want to connect to HyperChem from another complex application already containing a Tcl/Tk interpreter as an extension to that application.

2. You want your Tcl program to react to changes in HyperChem via HSV notifications. Notifications cannot be easily defined with internal Tcl/Tk scripting.
3. A new version of Tcl/Tk becomes available and it is not yet incorporated into HyperChem. The new version of Tcl/Tk has enhancements that are necessary to you.
4. The internal implementation of Tcl/Tk does not perform correctly for some non-regular scripts or you encounter a devastating bug.

Hypercube has implemented a Tcl/Tk extension "package" (called the THAPI package) that you can load into a standard Tcl/Tk interpreter and use to communicate with HyperChem through regular HAPI calls. This chapter describes this extension.

Invoking External Tcl/Tk

The external copy of Tcl/Tk is invoked in Microsoft Windows by executing the interpreter program, a WIndows SHell called Wish. Associated with Version 4.1 of Tk, this program is called *wish41* and is included in your program directory along with your executable of HyperChem.

Executing `wish41.exe` gives a Console window, shown below after executing "?" to see all the possible Tcl/Tk commands.

In addition to this console window, into which you type Tcl/Tk commands, you obtain another window where Tk widgets get placed after you request them from the Console window.

The THAPI package

It is possible to add new functions, called packages, to the external Tcl/Tk interpreter without having to recompile Tcl/Tk. The set of commands that can be added to an external copy of Tcl/Tk are part of the Tcl Hyperchem Application Programming Interface (THAPI). There are 12 commands in total and they are all contained in a Dynamic Link Library (DLL) called `THAPI.DLL`. These commands are essentially a subset of the HAPI calls from HAPI.DLL.

To obtain these new commands in Tcl/Tk you use the Tcl *load* command giving it the file name of the appropriate package DLL. Thus, to augment Tcl/Tk and embed all the HyperChem API calls, you should just execute the following Tcl command in the Console window,

```
load thapi
```

If the `THAPI.DLL` is not available in the current path nor in Windows directory, you must specify a full path to the file. Remember that Tcl comes from the UNIX world and you must type a slash, ”/”, rather than a back slash. “\” as the file folder separator in the Console window.

Commands

THAPI, based on the full HAPI interface of the last chapters of this book, defines a dozen new Tcl/Tk commands that enable you to call the most important features of the HyperChem Application Programming Interface from your Tcl/Tk external program. Further details on these commands is available in conjunction with a description of the HAPI calls in Appendix C. THAPI is a subset of HAPI.

The THAPI commands divide up into commands associated with connecting to HyperChem, the execution of Hcl commands, a utility copy command, HSV notifications, time-outs, and error processing.

The THAPI commands, which are case sensitive, are:

hcConnect <instance>

This command connects a Tcl/Tk program to HyperChem so that the other THAPI commands can be executed. It must be called before any other THAPI commands and after the command loading THAPI. The argument is optional but can be used to connect to a specific instance of HyperChem when multiple instances exist simultaneously.

hcDisconnect

This command disconnects the Tcl/Tk program from HyperChem.

hcExec hcl_script_command

This command passes its argument to HyperChem as a normal Hcl script command. The argument may need to be enclosed in quotes if it itself has arguments.

hcQuery hsv

This command queries HyperChem for the value of an HSV and returns it as a string

hcCopy source_file desination_file

This command copies a file.

hcNotifyStart hsv

This command requests a notification of the HSV corresponding to the argument. If the HSV changes in HyperChem its new value will sent to the Tcl/Tk script. The new value can be made available to the script via hcGetNotifyData. No call-back routines are available in scripts so that the execution of hcGetNotifyData must be periodically scheduled via a Tcl command like *after* (see the monitor example).

hcNotifyStop hsv

This command requests that the notification of changes in an HSV be terminated.

hcGetNotifyData notification_data

This command will place the result of the first notification from an internal buffer into the argument, *notification_data*. The command returns the HSV corresponding to the original notification (or NULL) so that you can check whether a notification has happened or not and whether it is the right one.

hcSetTimeouts exec_timeout query_timeout rest_timeout

This command controls the time-out for interaction with HyperChem if the default values (65 seconds) is inappropriate. The exec_timeout is the time-out for hcExec commands, the query_timeout is the time-out for hcQuery commands, and the rest_timeout is the time-out for the remaining Hcl commands, such as for notifications.

hcLastError error_text

This command places text describing the last error in error_text. It returns the following values:

errNO_ERROR = 0	No error
errFATAL = 1	Fatal error of unknown origin
errNON_FATAL = 2	Non fatal error of unknown origin

hcSetErrorAction action_flag

This command sets the behavior flag that will be used, upon recognition of an error. The arguments are as follows:

errACTION_NO = 0	No action on any error

errACTION_MESS_BOX = 16	Display message box with error message
errACTION_DISCONNECT = 32	Disconnect from HyperChem
errACTION_EXIT = 64	Immediately exit from application
errDDE_REP = 1	Report low-level DDE errors
errDDE_NO_REP = 2	Do not report low-level DDE errors

hcGetErrorAction

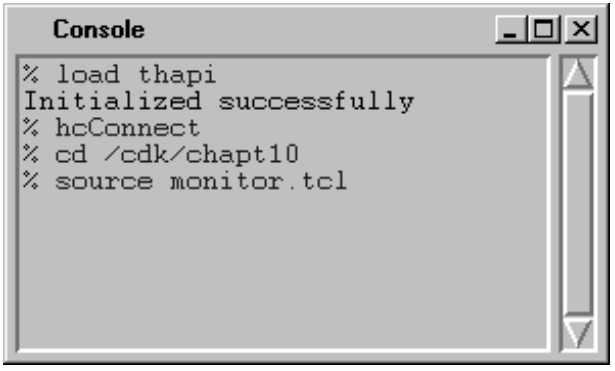
This command retrieves the behavior flag showing how errors will be acted upon. The values returned are the same as the values set by *hcSetErrorAction*.

A Notification Example

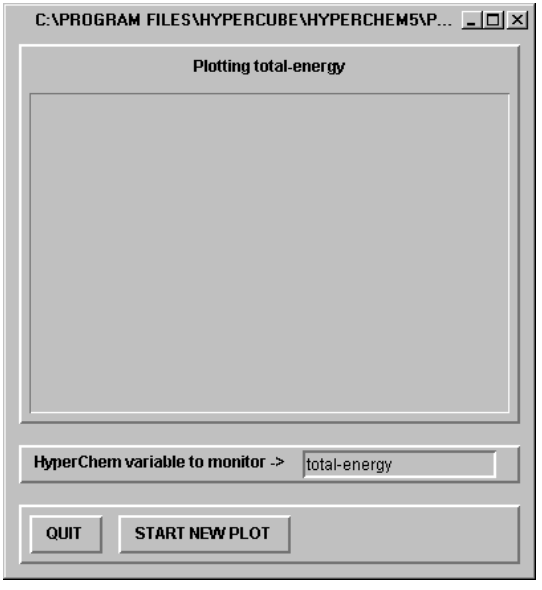
Notification is a powerful capability associated with HyperChem. It allows you to request a live link to HyperChem such that you are notified of any change in an indicated variable or data structure. This allows many capabilities that would not be possible otherwise. In particular, this makes it possible to have very intimate connections between your custom capability and the "guts" of HyperChem without getting into source code details. It is not required to know the intimate details of how HyperChem operates but only that it must be operating in certain ways. Thus, for example, one would know that at the heart of optimizations are energy and gradient changes without worrying about what specific algorithm is being used by HyperChem. You can just ask to monitor these changes without having to dive into code, algorithms, etc.

Most external Tcl scripts are identical with internal Tcl scripts. A notable exception is that notifications cannot be processed by internal Tcl scripts (HyperChem does not send messages to itself!). Notifications are normally associated with external programs like Microsoft Excel, Word or Visual Basic or external applications build in C, C++, and Fortan. If you intend to build software in Tcl that requires notification, you will need to do it with external Tcl/Tk. The example of this section is one that monitors and plots values of any HSV that is of potential interest to you. A specific example would be to monitor the energy or rms gradient of a structure optimization.

The Tcl script is executed from the Console as follows:



The *source* command begins the execution of `*.tcl` file as the next command. In this case it is `monitor.tcl` which sets up a Tk window that look as follows:



The Tcl code for setting up this window can be investigated by looking at the file installed from the CD-ROM but here we want to focus on the code associated with the notifications. The code behind the button, “Start New Plot” leaving out everything unessential to the notifications is,

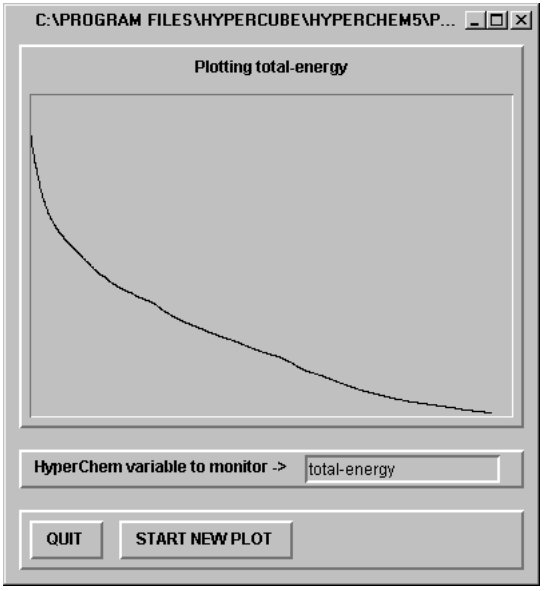
```
button .bts.plot -text "START NEW PLOT" -command {

            if {$Prevhsv != -1} {
               hcNotifyStop $Prevhsv
            }
            hcNotifyStart $hsv
            set Prevhsv $hsv

            if {$IsMonitor == -1} {
               monitor
               set IsMonitor 1
            }

            }
```

When the button is pushed, any previous notifications are first cancelled. Then a new notification is requested for the new HSV which shows in the window as total-energy and which is stored in the value, $hsv. A call is then made to a proc called monitor which will do the monitoring.

```
proc monitor {} {
global hsv interval
   set name [hcGetNotifyData a]
   if { $name == $hsv } {

         <TAKE $A, THE TOTAL ENERGY POINT AND SAVE FOR PLOTTING>
      }
# monitor restarts after plotting
after $interval [list monitor]
}
```

The code associated with collecting and plotting the notification data has been abstracted away. The monitoring is done by executing the command, *hcGetNotifyData*., to see if there really is any notification data available, i.e. the command returns the name of the HSV being monitored. If there was a notification, it is dealt with. If not, or there was data and it has been dealt with, a call is made to re-schedule a return to monitor after $interval milliseconds. In general, this means the process now goes to sleep and wakes up later to see if any notifications have arrived.

If you had connected the monitor example to HyperChem and then performed an optimization, the graph displayed in the TK window might look something like the following:


C:\PROGRAM FILES\HYPERCUBE\HYPERCHEM5\P...
Plotting total-energy
HyperChem variable to monitor ->
total-energy
QUIT
START NEW PLOT

Chapter 11

The HAPI Interface to HyperChem

Introduction

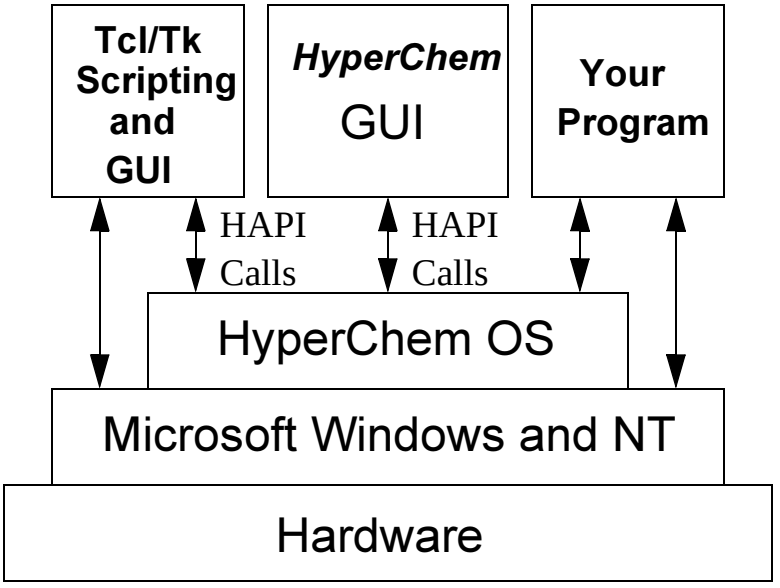
This chapter describes the HyperChem Application Programming Interface (HAPI or HyperChem API), the system library developed by Hypercube to simplify the task of communication with HyperChem. Instead of sending, posting and processing Windows DDE messages, an application makes HAPI calls and uses a set of functions provided by the API to manage the conversation between HyperChem and your external program. This is a high-level replacement for communicating via a lower-level DDE call.

The HyperChem API performs the task of packing and unpacking the data coming to and from messages so that arguments to HAPI calls can be in a form that you are familiar with and calls to the HAPI library become a simple extension to normal C or Fortran programming. The sophisticated error handling built into the HyperChem API guarantees robustness of the communication. While it is possible for applications to communicate with HyperChem without HAPI calls (as described in chapters 8 and 9, for example), it is not the recommended procedure. The HAPI library is meant for users intending to write their own sophisticated interfaces or back ends to the HyperChem front end. Moreover, some applications, such as most Fortran programs running under Windows 95 and NT, cannot easily interface to HyperChem without the HyperChem API.

Another reason for using the set of HAPI calls rather than DDE is the compatibility issue. There are indications that DDE may be replaced by another communication paradigm (OLE Automation) in future versions of Microsoft's operating systems. It thus might happen that Hypercube would need to change its underlying communication protocol from DDE to another messaging system in future releases of its HyperChem core product. Applications using the HAPI calls will preserve compatibility, while those that use a lower-level DDE protocol might lose future compatibility.

Towards a Chemical Operating System

The richness of HyperChem, evident in its HyperChem Command Language (Hcl), its embedded Tcl/Tk interpreter, and its HAPI library of calls, is indicative of a direction Hypercube is making towards a Chemical Operating System (The HyperChem OS). A picture of this system is as follows:



In this picture we have divided HyperChem into the fundamental services (building molecules, performing calculations, visualizing results, etc.) that it supplies to its normal users (and can provide to your external program) and the graphical user interface (GUI) that normal users use to get at these services. These services are all available by making HAPI calls to the HyperChem “Operating System”.

The Components

The components of the HyperChem Application Programming Interface are:

- hc.h - This is the header file for C and C++ interfaces that define each of the entry points for the HyperChem API.

- hsv.h - This header file defines an integer corresponding to each HSV and is needed if binary communication to HyperChem from C and C++ programs is to be used.
- hcload.c - This file contains the C code for the LoadHAPI routine that C and C++ programs should call to load the HAPI calls before using them. It generally is placed after the hc.h and hsv.h files in a C/ C++ program.
- HAPI.DLL - This is the DLL that must get loaded to access the HyperChem API.
- HAPI.LIB - This is the file that could be linked with your C, C++ or Fortran application as an alternative to calling LoadHAPI.
- hc.fi - The Fortran equivalent of hc.h as an include file to define the HAPI calls of your program.
- hsv.fi - The Fortran equivalent of hsv.h. This file is a necessary include file for your Fortran program if it is to attempt binary communication with HyperChem.
- hapi.bas - The Visual Basic equivalent of hc.h as an include file to define the HAPI calls of your program.
- hsv.bas - The Visual Basic equivalent of hsv.h. This file is a necessary include file for your Visual Basic program if it is to attempt binary communication with HyperChem.

The HAPI Calls

A complete and detailed documentation of each of the HAPI calls is given in Appendix C. Here we briefly list the calls and describe the kinds of calls that are in the library. These calls can be made from C, C++, Fortran, Visual, Basic, or external Tcl/Tk programs among others. In some situations, e.g. Tcl/Tk, the implemented set of calls is a subset of the full set.

In listing the calls below, we indicate their type and their arguments using a C-like syntax. Further details are again available in Appendix C. The calls for Fortran have the syntax, hfHAPICALL, the calls for Visual Basic have the syntax, hbHAPICALL, whereas the calls for other language situations begin with hc and have the syntax, hcHAPICALL.

Initialization and Termination

BOOL hcInitAPI (void)

This call is generally not needed in most contexts as initialization happens automatically.

BOOL hcConnect (LPSTR lszCmd)

This call connects to HyperChem. A non-null string as an argument can be used to connect to a specific instance of HyperChem.

BOOL hcDisconnect (void)

This call disconnects from HyperChem.

void hcExit(void)

This causes immediate termination of the calling application.

Discussion

The principal call of importance here is the one that connects to HyperChem. Prior to connecting, however, the HAPI library must have been loaded. See below how you should load the appropriate DLL or LIB file.

Text-based Basic Communication Calls

BOOL hcExecTxt (LPSTR script_cmd)

This sends an Hcl script command to HyperChem.

LPSTR hcQueryTxt (LPSTR var_name)

This queries an HSV in HyperChem.

Discussion

These are the text calls that send a command to HyperChem or perform a Read/Write of a HyperChem State Variable (HSV). They correspond to calls in the HyperChem Command Language (Hcl), i.e. they involve either a *menu invocation*, e.g. menu-file-open, a *direct command*, e.g. do-molecular - dynamics, an *HSV write*, e.g. window-color green, or an *HSV read*, e.g. win-

dow-color ?. The reading of an HSV is performed by the call, hcQueryTxt, while the other Hcl script commands are invoked by hcExecTxt.

Binary-based Basic Communication Calls

BOOL hcExecBin (int cmd, LPV args, DWORD args_length)

This is the binary form for sending an Hcl script command to HyperChem.

LPV hcQueryBin(int hsv, int indx1, int indx2, int* length)

This is the binary form for querying an HSV in HyperChem.

Discussion

The HyperChem Command Language is basically a text form for communicating and exchanging data with HyperChem. For completeness and for efficiency there is an equivalent binary form for all the Hcl script commands. These can be much more effective and simpler to use in languages like Fortran where one has to use character data to invoke normal script commands. See the programming examples for examples of both text and binary based communication with HyperChem.

With text based communication you use strings, like “window-color” and “do-molecular-dynamics” to denote the HSV of interest or the direct command that you want to invoke. With binary versions of hcExec and hcQuery you use an integer to denote the operation. The name of this integer has the syntax, window_color or do_molecular_dynamics, for example, where an underscore replaces the hyphen or minus sign of a Hcl text string. These integers are defined in hsv.h, hsv.fi, and hsv.bas for C and C++, Fortran, and Visual Basic applications. Only text-based communication is available for Tcl/Tk.

Binary Format

The binary form of a HAPI call results in a binary message being sent to HyperChem rather than a simple text message. The format of these binary messages is

Length*	Binary Code	Arguments**
4 bytes	4 bytes	n-bytes

*To make the binary message different from a text message it is required to set the highest bit of the first field to 1.

** Arguments are also coded as binary data.

The binary code for each command is the 4-byte integer number found in hsv.h, hsv.fi, or hsv.bas. The codes may change between different versions of HyperChem. Three utility Tcl scripts are included on the HyperChem CD-ROM, called `cgenhsv.tcl`, `fgenhsv.tcl`, and `bgenhsv.tcl` that are capable of generating the correct hsv.h, hsv.fi, and hsv.bas files for the version of HyperChem that you are using.

To issue a binary command the user uses, for example:

```
result=hcExecBin(hsv,arg,length);
```

in a C/C++ program, or

```
result=hfExecBin(hsv,arg,length)
```

in a Fortran program.

Binary-based Get Integer Calls

int hcGetInt (int hsv)

This call gets the binary value of an integer HSV.

int hcGetIntVec(int hsv, int* buff, int max_length)

This call gets all the binary values of an integer vector HSV into a buffer.

int hcGetIntArr (int hsv, int* buff, int max_length)

This call gets all the binary values of an integer array HSV into a buffer.

int hcGetIntVecElm (int hsv, int index)

This call gets a single binary element of an integer vector HSV.

int hcGetIntArrElm (int hsv, int atom_index, int mol_index)

This call gets a single binary element of an integer array HSV.

Discussion

These calls are specialized forms of hcQueryBin specific to integer variables, vectors, and arrays. One form gets the whole vector or array at once while the other form (Elm) gets only a single element at once.

Binary-based Get Real Calls

double hcGetReal (int hsv)

This call gets the binary value of a real HSV.

int hcGetRealVec(int hsv, double* buff, int max_length)

This call gets all the binary values of a real vector HSV into a buffer.

int hcGetRealArr (int hsv, double* buff, int max_length)

This call gets all the binary values of a real array HSV into a buffer.

double hcGetRealVecElm (int hsv, int index)

This call gets a single binary element of a real vector HSV.

double hcGetRealArrElm (int hsv, int atom_index, int mol_index)

This call gets a single binary element of a real array HSV.

int hcGetRealVecXYZ (int hsv, index, double* x, double* y, double* z)

This call gets the three real values of a single element of a real array HSV. The three real values are normally the Cartesian components of a real variable such as a dipole moment, etc.

int hcGetRealArrXYZ (int hsv, int atom_index, int mol_index, double* x, double* y, double* z)

This call gets the three real values of a single element of a real array HSV. The three real values are normally the Cartesian components of a real variable such as a coordinate, etc.

Discussion

These calls are specialized forms of hcQueryBin specific to real variables, vectors, and arrays. One form gets the whole vector or array at once while the other form (Elm) gets only a single element at once. The XYZ forms are specific to Cartesian values.

Binary-based Get String Calls

int hcGetStr (int hsv, char* buff, int max_length)

This call gets a string corresponding to all values of the HSV.

int hcGetStrVecElm (int hsv, int index, char* buff, int max_length)

This calls gets a string corresponding to a particular element of a vector.

int hcGetStrArrElm (int hsv, int atom_index, int mol_index, char* buff, int max_length)

This call gets a string corresponding to a particular element of an array.

Discussion

These calls correspond very closely to the text based calls in that they convey arguments as a string. However the HSV is represented as a binary integer not as a string.

Binary-based Set Integer Calls

int hcSetInt (int hsv, int value)

This call sets the binary value of an integer HSV.

int hcSetIntVec(int hsv, int* buff, int length)

This call sets length binary values of an integer vector HSV into a buffer.

int hcSetIntArr (int hsv, int* buff, int max_length)

This call sets length binary values of an integer array HSV into a buffer.

int hcSetIntVecElm (int hsv, int index, int value)

This call sets a single binary element of an integer vector HSV.

int hcSetIntArrElm (int hsv, int atom_index, int mol_index, int value)

This call sets a single binary element of an integer array HSV.

Discussion

These calls are specialized forms of hcExecBin specific to writing HSV's for integer variables, vectors, and arrays. One form sets the whole vector or array at once while the other form (Elm) sets only a single element at once.

Binary-based Set Real Calls

int hcSetReal (int hsv, double value)

This call sets the binary value of a real HSV.

int hcSetRealVec(int hsv, double* buff, int length)

This call sets length binary values of a real vector HSV into a buffer.

int hcSetRealArr (int hsv, double* buff, int length)

This call sets length binary values of a real array HSV into a buffer.

int hcSetRealVecElm (int hsv, int index, double value)

This call sets a single binary element of a real vector HSV.

int hcSetRealArrElm (int hsv, int atom_index, int mol_index, double value)

This call sets a single binary element of a real array HSV.

int hcSetRealVecXYZ (int hsv, index, double x, double y, double z)

This call sets the three real values of a single element of a real array HSV. The three real values are normally the Cartesian components of a real variable such as a dipole moment, etc.

int hcSetRealArrXYZ (int hsv, int atom_index, int mol_index, double x, double y, double z)

This call sets the three real values of a single element of a real array HSV. The three real values are normally the Cartesian components of a real variable such as a coordinate, etc.

Discussion

These calls are specialized forms of hcExecBin specific to writing HSV's for real variables, vectors, and arrays. One form sets the whole vector or array at once while the other form (Elm) sets only a single element at once. The XYZ forms are specific to Cartesian values.

Binary-based Set String Calls

int hcSetStr (int hsv, char* string)

This call sets an HSV to the value of a string.

int hcSetStrVecElm (int hsv, int index, char* string)

This call sets a particular element of an HSV vector to a string.

int hcSetArrElm (int hsv, int atom_index, int mol_index, char* string)

This call sets a particular element of an HSV array to a string.

Discussion

These calls correspond very closely to the text based calls in that they convey arguments as a string. However the HSV is represented as a binary integer not as a string.

Get and Set Blocks

int hcGetBlock (int hsv, char* buff, int max_length)

This call gets all the data, irrespective of type, corresponding to an HSV.

int hcSetBlock (unt hsv, char* buff, int length)

This call sets an HSV, irrespective of its type from a buffer.

Discussion

These calls do block copies of data associated with an HSV of any type.

Notification Calls

int hcNotifyStart (LPSTR hsv)

This call requests a notification for an HSV.

int hcNotifyStop (LPSTR hsv)

This call terminates a request for notification of an HSV.

int hcNotifySetup (PFNB pCallBack, int NotifyWithText)

This call establishes how notifications are to be handled.

int hcNotifyDataAvail (void)

This call determines whether a notification is available.l

int hcGetNotifyData (char* hsv, char* buff, int max_length)

This call gets the data associated with a notification on an HSV.

Discussion

These calls are associated with notifications.

Memory Allocation

void * hcAlloc (size_t, n_bytes)

This allocates memory associated with the HAPI but is not recommended to replace the normal user memory allocation routines.

hcFree (void* pointer)

This frees memory allocated with hcAlloc or after processing data from hcQueryTxt and hcQueryBin which allocate memory for the result of the query.

Discussion

These calls are for memory allocation and deallocation associated with the HAPI.

Auxiliary Calls

void hcShowMessage (LPSTR message)

This call displays a message box with the message provided.

void hcSetTimeouts (int ExecTimeout, int QueryTimeout, int OtherTimeout)

This call sets timeouts for hcExec, hcQuery, and other HAPI calls.

int hcLastError (char* LastErr)

This call enquires about the last error.

int hcGetErrorAction (void)

This call asks how errors are currently being handled.

void hcSetErrorAction (int err)

This call sets how errors are to be handled.

Discussion

Further details on these calls are available in Appendix C.

The HAPI Dynamic Link Library (HAPI.DLL)

The HyperChem Application Programming Interface (HAPI), forms a set of system calls that can be utilized by an application to perform certain tasks in conjunction with HyperChem. It is somewhat analogous to the Microsoft

Windows API and is implemented in a similar way, through a dynamic Link Library (DLL).

The whole Microsoft Windows operating system is seen by a user program as an API - the Microsoft Windows 32-bit API for Windows 95 and NT. For example, this means that a user writing code in C can utilize all Microsoft's system calls that form the API. In addition to the generic Windows API there are specific API's, such as the WINSOCK API - the library that defines TCP/IP communication for Windows applications. Usually, a Microsoft API is provided to users in a form of a DLL.

A Dynamic Link Library is specific to Microsoft Windows and NT and as a library has one important feature: it can be linked to the user program at run-time. In most cases, it can also be linked while the application is loaded into memory (load time). At run-time, however, it can be loaded any time during program execution, as well as unloaded when it is no longer required. This feature is of great importance, as it provides both users and developers a painless method of improving upon both the standard Microsoft API and an application specific API.

The HyperChem API (HAPI) has been developed as a 32-bit DLL that is compatible with Microsoft Windows 95, Microsoft NT and Microsoft's W32S 32-bit subsystem for Windows 3.1. Thus, it can be used with all the programming tools that call 32-bit DLLs including almost all modern C/C++ and Fortran compilers, 32-bit versions of Microsoft Visual Basic, and tools like the Tcl/Tk interpreter, Microsoft Excel, Borland's Delphi, etc.

How to use the HyperChem API

To make the HyperChem API available to your application, you must have the requisite header files, load the HAPI.DLL properly, and then make a proper connection of your program to HyperChem. Alternatively, it is possible to use a library file, HAPI.LIB, link it with your application and use load-time dynamic linking. Though it is possible for you to make use of he HyperChem API with a variety of compilers and development systems through intelligent use of these tools, inspecting our header files, etc., we believe it makes sense to illustrate the tools very explicitly with a very small variety of standard environments. Thus, we have chosen to provide standard interface source code along with the CDK so that at least for certain environments an interface to HyperChem becomes very straight-forward. We have chosen the following to illustrate the HyperChem API:

- - Microsoft 32-bit C/C++ compilers (from Visual C++ 4.0)

- - Microsoft Power Station Fortran compiler (version 4.0 and higher)
- - Microsoft Visual Basic 4.0
- - Tcl/Tk implemented as a 32-bit “wish41” Windows application

Nevertheless, if you must interface to the HyperChem API from another type of application, you can use the information in this section combined with each function’s header information to write your own code to call the API. Such a situation might happen when you want to use other tools such as Borland Delhi, for example.

Accessing the HyperChem API from C/C++ code

Run-Time Dynamic Linking

The method for accessing a function in the HAPI depends on the programming language used. For C/C++ programs, the easiest method is to include the provided `hcload.c` source code just after including the main header file, `hc.h`. The typical sequence is as follows:

```
#include "hc.h"
#include "hcload.c"
```

Alternatively, instead of including `hcload.c` into each module of the larger application it may be convenient to link the application with `hcload.obj` during a static linkage phase. In this case the program only needs:

```
#include "hc.h"
```

The corresponding ‘*makefile*’ (or project workspace) would have the form:

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#+    EXAMPLE make file    +
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

userapp.exe : userapp.obj hcload.obj
     $(LINKER) $(GUIFLAGS) -OUT:userapp.exe userapp.obj hcload.obj
$(GUILIBS)

userapp.obj : userapp.c
     $(CC) $(CFLAGS) step0.c

hcload.obj : hcload.c
     $(CC) $(CFLAGS) hcload.c
```

In the environment of the Microsoft Developer Studio, the corresponding action would be to insert `hcload.c` as another source file among the user's files, i.e. by using the menu item, <Insert/Files into Project...>.

Within your source code one of the first things to do is to dynamically load, or activate the library. Your program has to call the LoadHAPI function defined in `hcload.c`. A typical sequence would be:

```
/* loading HAPI.DLL   */
      if (!LoadHAPI("hapi.dll")) {
      MessageBox(hwnd,"Error loading HAPI.DLL",
      "Error",MB_OK | MB_ICONSTOP);
      exit(0);
      }
```

The *LoadHAPI* call takes one parameter which is a file name, HAPI.DLL. Note that the DLL can be placed in the local directory, but the most appropriate place is the main Windows system directory (usually C: \WINDOWS). *LoadHAPI* activates each of the functions in the DLL by making repetitive calls to *GetProcAddress*, the function provided by Microsoft to get the address of the corresponding function in the DLL:

```
hcExecTxt=(T_hcExecTxt*)GetProcAddress(hinst,"hcExecTxt");
if ( hcExecTxt == NULL) res=_hcLoadError("hcExecTxt",szN);
```

This method utilized by LoadHAPI is called Run-Time Dynamic Linking.

Load-Time Dynamic Linking

There is another method to link HAPI with a user application. It uses Load-Time Dynamic Linking and requires direct linking of the user program with the library file (HAPI.LIB) provided with the CDK. This file contains no code but contains all of the API function headers and proper initialization code. You can use load-time linking by inserting HAPI.LIB as one of the libraries into a project, i.e. by the menu command, < Insert/Files into Project>, or by writing an appropriate makefile when working outside of the Microsoft Developer Studio (using a DOS command shell for compilation).

In either cases the application still needs:

```
#include "hc.h"
```

However, with load-time linking, all the HAPI functions are available for calling as soon as the application starts. There is no need for the *LoadHAPI* call.

If you are making binary calls, it is necessary to have:

```
#include "hsv.h"
```

Accessing the HyperChem API from Fortran code

To use the HAPI calls from Fortran programs you have to translate Fortran-style calls into the appropriate system calls to the HAPI. With Microsoft Power Station Fortran it suffices to just include the file, `hc.fi`, provided with the CDK and then statically link in HAPI.LIB. The code in `hc.fi` has to be included in all functions and subroutines that use HAPI calls. For example:

```
   subroutine ExeTest
     character*100 cmd
     common /flags/flagfile,fconnected
     logical res,flagfile,fconnected
     include "hc.fi"

1    write(*,*)' Script command to execute (0 - returns) ->'
     read(*,'(A)')cmd
     if (cmd.eq.'0') return
     res=hfExecTxt(cmd)
     write(*,*)' hfQueryTxt returns: ',res
     goto 1

   end
```

What does the source inside `hc.fi` look like and do? It provides an INTERFACE (a Fortran capability of Microsoft Fortran Power Station), so the Fortran compiler can type-check all function parameters and issue the proper calling sequences. The source looks as follows:

```
INTERFACE
c---------------------------------------------------------------------
c This is a Microsoft Fortran Power Station 4 Interface to CDK API
c---------------------------------------------------------------------
c---------------------------------------------------------------------
c Initialization & termination functions
c---------------------------------------------------------------------
   logical function hfInitAPI()
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hcInitAPI@0' :: hfInitAPI
   end function hfInitAPI
```

```
     logical function hfConnect(init_string)
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hfConnect@8' :: hfConnect
      character*(*) init_string
   end function hfConnect

     logical function hfDisconnect()
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hcDisconnect@0' :: hfDisconnect
   end function hfDisconnect

   subroutine hfExit()
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hcExit' :: hfExit
   end subroutine hfExit
c--------------------------------------------------------------------
c--------------------------------------------------------------------
c Text Query & Execute functions
c--------------------------------------------------------------------

   logical function hfExecTxt(script_cmd)
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hfExecTxt@8' :: hfExecTxt
      character*(*) script_cmd
!ms$ATTRIBUTES REFERENCE :: script_cmd
   end function hfExecTxt

   logical function hfQueryTxt(var_name, res)
      character*(*) var_name,res
!MS$ATTRIBUTES reference :: var_name
!MS$ATTRIBUTES reference :: res
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hfQueryTxt@16' :: hfQueryTxt
   end function hfQueryTxt

.....................................

c--------------------------------------------------------------------
   integer function hfLastError(error)
!ms$ATTRIBUTES DLLIMPORT,ALIAS: '_hcLastError@4' :: hfLastError
!ms$ATTRIBUTES C,REFERENCE :: error
    character *(*) error
   end function hfLastError

   integer function hfGetErrorAction()
!ms$ATTRIBUTES
DLLIMPORT,ALIAS:'_hcGetErrorAction@0'::hfGetErrorAction
   end function hfGetErrorAction
```

```
   subroutine hfSetErrorAction(action)
!ms$ATTRIBUTES
DLLIMPORT,ALIAS:'_hcSetErrorAction@4'::hfSetErrorAction
!ms$ATTRIBUTES VALUE :: action
     integer action

   end subroutine hfSetErrorAction

c-----------------------------------------------------------------
c end of hc.fi
c-----------------------------------------------------------------
   END INTERFACE
```

If you are making binary calls, it is also necessary to have:

```
include "hsv.fi"
```

Accessing the HyperChem API from Visual Basic Code

While it is relatively straight forward for Visual Basic programs to communicate with HyperChem via DDE as described in Chapter 9, it may be desirable to use the HyperChem API and HAPI.DLL from Visual Basic for similar reasons to that described above for C, C++, and Fortran. It is possible to do so with the principal requirement being the need to have two files, `hapi.bas` and `hsv.bas` that define the interface to the HAPI.DLL The following is a portion of the `hapi.bas` file,

```
Attribute VB_Name = "Module1"

Declare Function hbInitAPI Lib "hapi.dll" Alias "hcInitAPI" () As Long
Declare Function hbConnect Lib "hapi.dll" Alias "hcConnect" (ByVal
                           command As String) As Long
Declare Function hbDisconnect Lib "hapi.dll" Alias "hcDisconnect" ()
                           As Long
Declare Sub  hbExit Lib "hapi.dll" Alias "hcExit" ()
Declare Function hbExecTxt Lib "hapi.dll" Alias "hcExecTxt" (ByVal
                           script_cmd as string) As Long

Declare Function hbExecBin Lib "hapi.dll" Alias "hcExecBin" (ByVal
                           cmd as Long, ByRef args as Long,
                           ByVal args_length as Long ) As Long
```

```
Declare Function hcQueryTxt Lib "hapi.dll" (ByVal command As String)
                            As String
Declare Function hfQueryBinLib "hapi.dll" (ByVal var,indx1,
                            indx2 as integer, ByRef result as integer,
                            ByRef cbL as integer) As Integer
```

The Visual Basic application, DLA, which is on the HyperChem CD-ROM, is an example of a Visual Basic application that uses HAPI calls.

Accessing the HyperChem API from Tcl/Tk code

A description of this interface is given in Chapter 10. Only text-based communication is supported. Notifications are processed by the Notification Agent.

Considerations for Console-based Applications

A "Console application" is a new application type available for Windows 95 and Windows NT operating systems. It is an implementation of the popular text-based terminal environment well known in the UNIX world. Such a program uses a regular *main()* function as an entry point, rather than *WinMain()*. Instead of communicating with the user through a GUI , the console application simply *writes* to the screen using the regular *printf* call and *reads* using the *scanf* call. This is for C code. In Fortran the familiar *write, print,* and *read* calls are available.

But the real differences between GUI-based and Console-based applications are much larger than just input/output. All regular Windows application are *event-driven* applications, where all processing is performed by reacting to incoming messages or events, generated by the operating system. This makes writing Windows programs a difficult task for most traditional programmers with a so called top-down or serialized view of programming. This represents a new paradigm for programming that is not in the experience of most UNIX-style programmers.

The console interface makes the conversion of programs from UNIX to Windows easy and painless. Now, under Windows 95 and NT, almost all older UNIX C and FORTRAN programs can be compiled without problems and without severe modifications to the code. The underlying POSIX-compatible system calls layer, available for Windows NT, makes the porting to NT of a large number of "legacy" applications possible.

However, because console applications are not event-driven, Microsoft decided to insulate them from certain features of Windows. Particularly, there is no possibility for an application to be called via regular DDE callback functions. In practice, DDE communication can be used by the console application, as long as there is no need for the application *to be called* through a DDE Callback function. However, this "callback" feature is mandatory to receive notifications from HyperChem about the change in an HSV variable.

The Notification Agent

To work around this, HyperCube, has implemented "The Notification Agent". The agent is another thread of execution (both Win95 & NT are real multithreading applications) that does the following:

- Starts a regular Windows procedure (analogous to *WinMain*) by registering the window class and defining a *WndProc*-type of function that can process all windows messages.
- The agent then opens its own DDE communication with HyperChem.
- The agent registers and processes notifications and allocates its own buffers when necessary.
- Each notification results in buffer allocation up to the limit of memory resources.
- The notification buffers are freed when a user copies their contents into their own memory.
- The user application access the buffers which form very simple linked list using only two functions:

From a C Program:

```
DWORD _stdcall hcNotifyDataAvail()

DWORD _stdcall hcGetNotifyData(char* name, char *buffer, DWORD
MaxBuffLength)
```

From a Fortran program:

```
integer function hfNotifyDataAvail()
integer function hfGetNotifyData(name,result,res_length)
```

- The first informs the caller if there is some data that has came in as a notification message. The second copies the top-most buffer into the user program, deallocating the buffer.
- The proper accessing by the notification agent and your user program of a critical section of storage (the buffer area and control area) is handled by use of semaphores, making the whole solution very robust. Tests made under both Windows 95 and NT have shown the proper handling of data without any loss of incoming messages, irrespective of their size.

The notification agent is useful also for those applications that cannot define a callback (like a regular Tcl/Tk scripting application). It is also the method used by HyperChem to avoid an event-driven programming paradigm in console applications, particularly.

Examples of HAPI Calls

C, C++

Text-based

```
int result;
result = hcExecTxt("do-molecular-dynamics");
result = hcExecTxt("menu-file-open");
result = hcExecTxt ("dipole moment 2.5");
```

Binary-based

```
int result;
double value;
value = 2.5;
result = hcExecBin (do_molecular_dynamics);
result = hcExecBin (menu_file_open);
result = hcExecBin (dipole_moment, &value, 8);
```

Fortran

Text-based

```
integer result
result = hfExecTxt ('do_molecular_dynamics')
```

```
result = hfExecTxt ('menu-file-open')
result = hfExecTxt ('dipole-moment 2.5')
```

Binary-based

```
integer result
double precision value
value = 2.5
result = hfExecBin (do_molecular_dynamics)
result = hfExecBin (menu_file_open)
```

Visual Basic

Text-based

```
Dim result As Long
result = hbExecTxt ("do_molecular_dynamics")
result = hbExecTxt ("menu-file-open")
result = hbExecTxt ("dipole-moment 2.5")
```

Binary-based

```
Dim result As Long
Dim value As Double
value = 2.5
result = hbExecBin (do_molecular_dynamics)
result = hbExecBin (menu_file_open)
result = hbExecBin (dipole_moment, value, 8)
```

Chapter 12

Development Using the Windows API

Introduction

This chapter describes the development of “Standard” Windows programs that interface with HyperChem. By standard we mean that these programs are written in C and use the lower-level approach of calling the Windows Application Programming Interface (API) directly. Such programs are said to be developed with the System Developer Kit (SDK). This contrasts with the next chapter which describes development of Windows programs using C++ and the Microsoft Foundation Classes. The SDK is in some sense the *hard way* to do Windows programming but it is also the most basic and flexible way to build a Windows program and most of the commercial Windows applications you will encounter have been written in the following fashion.

Microsoft Development Tools

The interfacing examples of these next few chapters assume that you have access to certain Windows development tools. While much of what we describe is generic to a selection of compilers and programming environments, we have specifically used Microsoft’s tools in all of our examples. Thus, this chapter and the next make use of the C and C++ compilers of Microsoft Visual C++, version 4.0, for 32-bit development. We have used this compiler for HyperChem Release 5.0. The information needed to interface your program to HyperChem is fundamentally independent of the specific Windows tools used, but there may be small changes necessary to adapt to your specific tool set if you are not using Visual C++ 4.0.

Programming Assistance

This manual is certainly not a programming manual for Windows but you may need one. With the entering of any new area of endeavor, it is comforting to have the right information and assistance from experts. If you are new to

the type of programming illustrated in this chapter, we very strongly recommend that you get a copy of the following book,

Programming Windows

Charles Petzold and Paul Yao

ISBN 1-55615-676-6

Microsoft Press, 1996

This is a new version of the programming classic by Charles Petzold. Since Windows NT programming is very similar to that for Windows 95, it is an appropriate book for NT development as well. There is an assumption in this manual that you are probably a UNIX programmer. The Petzold book is an excellent way for you to approach programming for Windows and NT and we highly recommend it. A great deal has been written about programming for Windows and there exist other fine reference books as well.

Language

This chapter uses the C language and the next chapter uses C++, rather than Fortran, which is generally much more familiar to scientific programmers and may be your language of choice. In Chapter 14 we describe how to write so-called *console* programs in Fortran that can be interfaced to HyperChem. However, for writing a *normal* Windows program that has a graphical user interface, visualization, etc., Fortran is somewhat problematic and C or C++ are much to be preferred for normal Windows programming. With the Chemist's Developer Kit, C, C++, and VB are the languages that we anticipate you will be using for true "Windows-like" development. Any of the four languages, C, C++, VB, and Fortran are appropriate for the "character-like" programming common to most large computational chemistry modules.

A First Example

The first programming example will again be our "Red and Green" example which is the equivalent in this manual to the common "Hello World" programming examples that you probably have seen elsewhere.

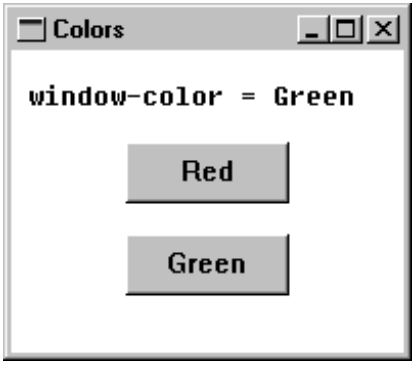
This example is referred to as Colors and will be in its own Colors directory from the HyperChem CD-ROM. It is assumed that you have installed Visual C++ 4.0 and that you are working from a DOS box in Windows 95. The first thing that you must do, as with other programming examples from this chapter, is to go to the proper directory and execute MSC.BAT to configure your environment for the Microsoft compiler.

1. Change to the COLORS directory
2. Type MSC.BAT

If you have any trouble with running out of environment space, click on the Properties tool in the DOS-box tool bar to increase the amount of environment memory. You are now ready to compile the `COLORS.C` file and create the executable. Type `NMAKE`. You should see your program being compiled and the executable `COLORS.EXE` being created. Make sure HyperChem is running and then execute `COLORS.EXE`. That is,

3. Type NMAKE
4. Make sure HyperChem is running
5. Type COLORS to execute COLORS.EXE

You could do this last step by typing `COLORS` in your DOS box or by double clicking on the Colors icon inside the Explorer or by having HyperChem execute the script `COLORS.SCR` which simply has in it the Hcl script command, *execute-HyperChem-client colors.exe*. You should then see the following Windows Application on the screen:



If you push the Red button, the HyperChem background window color will turn red. If you push the Green button, it will turn green. The text above the buttons indicates the current color of the HyperChem window, if its last change came from the external application, Colors. If you change the window from the <File/Preferences...> dialog box within HyperChem, the external application will not know about the change (a notification operation is necessary for this).

The code for this Windows application is shown below. It is a completely standard “boiler-plate” windows code except for the text shown in bold.

```
/*
Window Colors - SDK program to connect and talk to HyperChem
*/
#include <windows.h>
#include "hc.h"
char cmd_line[120];
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
   {
   static char szAppName[] = "Colors" ;
   HWND        hwnd ;
   MSG         msg ;
   WNDCLASSEX  wndclass ;
   int         windowx,windowy;
   wndclass.cbSize        = sizeof (wndclass) ;
   wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
   wndclass.lpfnWndProc   = WndProc ;
   wndclass.cbClsExtra    = 0 ;
   wndclass.cbWndExtra    = 0 ;
   wndclass.hInstance     = hInstance ;
   wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
   wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
   wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
   wndclass.lpszMenuName  = NULL ;
   wndclass.lpszClassName = szAppName ;
   wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
   RegisterClassEx (&wndclass) ;
   lstrcpy(cmd_line,szCmdLine);
   windowx=200;
   windowy=175;
   hwnd = CreateWindow (szAppName, "Colors",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        windowx,
                        windowy,
                        NULL, NULL, hInstance, NULL) ;
   ShowWindow (hwnd, iCmdShow) ;
   UpdateWindow (hwnd) ;
   while (GetMessage (&msg, NULL, 0, 0))
      {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
```

```
          }
     return msg.wParam ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
            LPARAM lParam)
   {
   static HWND  hwndButton[2] ;
   static RECT  rect ;
   static int   cxChar, cyChar ;
   HDC          hdc ;
   PAINTSTRUCT  ps ;
   TEXTMETRIC   tm ;
   static char  szBuff[200], szInitColor[50];
   char         *response;
   int          result;
   static int   connected=FALSE;
   switch (iMsg)
      {
         case WM_CREATE :
         hdc = GetDC (hwnd) ;
         SelectObject (hdc,GetStockObject(SYSTEM_FIXED_FONT));
         GetTextMetrics (hdc, &tm) ;
         cxChar = tm.tmAveCharWidth ;
         cyChar = tm.tmHeight + tm.tmExternalLeading ;
         ReleaseDC (hwnd, hdc) ;

         hwndButton[0]=CreateWindow("button","Red",
               WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
               7*cxChar, 3*cyChar, 10*cxChar, 2*cyChar,
               hwnd, (HMENU)0,
               ((LPCREATESTRUCT) lParam) -> hInstance,
               NULL) ;
         hwndButton[1]=CreateWindow("button","Green",
               WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
               7*cxChar, 6*cyChar, 10*cxChar, 2*cyChar,
               hwnd,(HMENU)1,
               ((LPCREATESTRUCT) lParam) -> hInstance,
               NULL) ;
         return 0;

         caseWM_SHOWWINDOW :
               if (!connected)
               {
```

```
        /* loading CDK API DLL   */
        if (!LoadHAPI("hapi.dll"))
        {
        MessageBox(hwnd,"Error loading CDK's API DLL !",
                    "Error", MB_OK | MB_ICONSTOP);
        exit(0);
        }
        /* connecting to HyperChem  */
        if (!hcConnect(cmd_line))
        {
        MessageBox(hwnd,"Error connecting with HyperChem !",
                    "Error", MB_OK | MB_ICONSTOP);
        exit(0);
        }
        else
        {
        connected=TRUE;
        }
        /* obtaining initial window-color from HyperChem */
        response=hcQueryTxt("window-color");
        lstrcpy(szInitColor,response);
        hcFree(response);
        wsprintf(szBuff,"%s",szInitColor);
        }
        return 0;

    case WM_PAINT :
          InvalidateRect (hwnd, &rect, TRUE) ;
          hdc = BeginPaint (hwnd, &ps) ;
          SelectObject(hdc,GetStockObject(SYSTEM_FIXED_FONT));
          SetBkMode (hdc, TRANSPARENT) ;
          TextOut(hdc,cxChar,cyChar,szBuff,lstrlen(szBuff)) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

    case WM_DRAWITEM :
    case WM_COMMAND :
          hdc = GetDC (hwnd) ;
          SelectObject(hdc,GetStockObject(SYSTEM_FIXED_FONT));
          switch (LOWORD (wParam)) {
             case 0 :
             wsprintf(szBuff,"window-color = Red");
             result=hcExecTxt(szBuff);
```

```
                break;
             case 1 :
                wsprintf(szBuff,"window-color = Green");
                result=hcExecTxt(szBuff);
                break;

             default : ;
          }
          /*TextOut (hdc, cxChar, cyChar,
                      szBuff,lstrlen(szBuff));*/
          ReleaseDC (hwnd, hdc) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          break ;

       case WM_DESTROY :

          result=hcExecTxt(szInitColor);
          result=hcDisconnect();
          PostQuitMessage (0) ;
          return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

The changes to a standard version of the WinMain routine, as used by all Windows programs, are very minimal. the only changes are to include a header file, hc.h, to change the size of the window to a smaller than normal value, to save the invoking command line for potential use, and to change the name of the main window to Colors.

The changes to WndProc, the callback procedure, are more extensive. This is the routine that contains the code you get to execute when certain events occur. It is, of course, very application dependent although all Windows programs have a similar basic outline.

Modification of a Molecule's Coordinates

The next example that we demonstrate and explain is one which has a very fundamental capability similar to many programs that you might wish to write. That is, this program gets a molecule and its coordinates from HyperChem, modifies these coordinates in some fashion and returns them to HyperChem for continuous display. A more elaborate example might extend this program to perform a geometry optimization, to execute a molecular dynam-

ics, trajectory, etc. Once again the modifications from a standard Windows program are denoted in bold face type.

```
/*
 * C-API Examples*********************************************
 * Rotation - program to demonstrate  *
 * modification of coordinates in the HyperChem workspace  *
*****************************************************************
*/
#include <windows.h>
#include <math.h>
#include <stdio.h>
#include "hc.h"
#include "hsv.h"
char cmd_line[100];
static char *Label[] = {"Step Size",
                        "Total Steps"};
#define _CW(l) cxChar*(lstrlen(l)+1)
#define MAX_BUFF 200
typedef struct _ATM_COORDS { double x,y,z; } ATM_COORDS;
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
   {
   static char szAppName[] = "Rotate" ;
   HWND        hwnd ;
   MSG         msg ;
   WNDCLASSEX  wndclass ;
   int windowx,windowy;
   wndclass.cbSize        = sizeof (wndclass) ;
   wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
   wndclass.lpfnWndProc   = WndProc ;
   wndclass.cbClsExtra    = 0 ;
   wndclass.cbWndExtra    = 0 ;
   wndclass.hInstance     = hInstance ;
   wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
   wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
   wndclass.hbrBackground=(HBRUSH)GetStockObject(COLOR_BACKGROUND);
   wndclass.lpszMenuName  = NULL ;
   wndclass.lpszClassName = szAppName ;
   wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
   RegisterClassEx (&wndclass) ;
   lstrcpy(cmd_line,szCmdLine);
```

```
   windowx=280;
   windowy=200;
   hwnd = CreateWindow (szAppName, "Rotate",
                  WS_OVERLAPPEDWINDOW,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  windowx,
                  windowy,
                  NULL, NULL, hInstance, NULL) ;
   ShowWindow (hwnd, iCmdShow) ;
   UpdateWindow (hwnd) ;
   while (GetMessage (&msg, NULL, 0, 0))
   {
   TranslateMessage (&msg) ;
   DispatchMessage (&msg) ;
   }
   return msg.wParam ;
   }

LRESULT CALLBACK WndProc(HWND hwnd,UINT iMsg,
                           WPARAM wParam,LPARAM lParam)
   {
   static HWND hwndButton[1],hwndLabel[2],hwndEdit[2] ;
   static RECT rect ;
   static int  cxChar, cyChar ;
   HDC         hdc ;
   TEXTMETRIC  tm ;
   HINSTANCE   hins;
   static char szBuff[MAX_BUFF],szInitColor[50];
   int         result,i;
   static int  connected=FALSE;
   static      ATM_COORDS *org_xyz,*new_xyz;
   static int  nMol,*nAtm,nAtmTot;
   double      dRot,rAng,rCos,rSin;
   int         nRot,ia;

   switch (iMsg)
   {
   case WM_CREATE :
      hdc = GetDC (hwnd) ;
      SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
      GetTextMetrics (hdc, &tm) ;
      cxChar = tm.tmAveCharWidth ;
      cyChar = tm.tmHeight + tm.tmExternalLeading ;
```

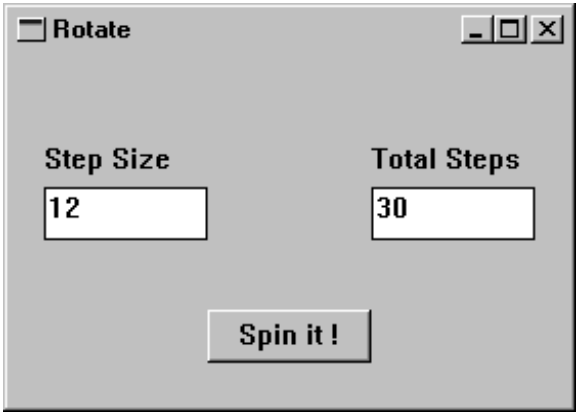
```
      ReleaseDC (hwnd, hdc) ;
      hins=((LPCREATESTRUCT) lParam) -> hInstance;

      hwndLabel[0]=CreateWindow("static",
                        Label[0],
                        WS_CHILD | WS_VISIBLE,
                        2*cxChar,
                        3*cyChar,
                        _CW(Label[0]),
                        2*cyChar,
                        hwnd,(HMENU)101, hins, NULL);
      hwndEdit[0]=CreateWindow("edit",
                        NULL,
                        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
                        2*cxChar,
                        (int)(4.5*cyChar),
                        10*cxChar,
                        7*cyChar/4,
                        hwnd,
                        (HMENU)200 ,hins, NULL);
      hwndLabel[1]=CreateWindow("static",
                        Label[1],
                        WS_CHILD | WS_VISIBLE,
                        22*cxChar,
                        3*cyChar,
                        _CW(Label[1]),
                        2*cyChar,
                        hwnd,(HMENU)102, hins, NULL);
      hwndEdit[1]=CreateWindow("edit",
                        NULL,
                        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
                        22*cxChar,
                        (int)(4.5*cyChar),
                        10*cxChar,
                        7*cyChar/4,
                        hwnd,(HMENU)201 ,hins, NULL);
      hwndButton[0]=CreateWindow("button",
                        "Spin it !",
                        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                        12*cxChar,
                        (int)(8.5*cyChar),
                    10*cxChar,
                    7*cyChar/4,
```

```
                    hwnd,(HMENU)0,hins, NULL) ;

   return 0;

case  WM_SHOWWINDOW :
   if (!connected) {
      /* loading CDK API DLL   */
      if (!LoadHAPI("hapi.dll")) {
      MessageBox(hwnd,
         "Error loading CDK's API DLL !",
          "Error", MB_OK | MB_ICONSTOP);
      exit(0);
      }
   /* connecting to HyperChem  */
   if (!hcConnect(cmd_line)) {
      MessageBox(hwnd,
         "Error while connecting with HyperChem !",
          "Error", MB_OK | MB_ICONSTOP);
      exit(0);
      } else {
      connected=TRUE;
      }
   /* obtaining coordinates from HyperChem */
   nMol=hcGetInt(molecule_count); // getting number of molecules
   if (nMol < 1) {
      MessageBox(hwnd,
      "There are no molecules to play with !",
       "Error", MB_OK | MB_ICONSTOP);
      hcDisconnect();
      exit(0);
      };
   nAtm=(int*)hcAlloc(nMol*sizeof(int));
   // getting vector specifing number of atoms in each
                                    // molecule
   if (!nAtm) {
      MessageBox(hwnd,
         "Memory allocation error",
         "Error", MB_OK | MB_ICONSTOP);
      hcDisconnect();
      exit(0);
      };
   if (!hcGetIntVec(atom_count,nAtm,nMol)) {
      MessageBox(hwnd,
```

```
            "Error while getting data from HyperChem !",
            "Error", MB_OK | MB_ICONSTOP);
         hcDisconnect();
         exit(0);
         };
      nAtmTot=0;
      for (i=0;i<nMol;i++) nAtmTot += nAtm[i];
      org_xyz=(ATM_COORDS*)hcAlloc(nAtmTot*sizeof(ATM_COORDS));
      new_xyz=(ATM_COORDS*)hcAlloc(nAtmTot*sizeof(ATM_COORDS));
      if ( (!org_xyz) || (!new_xyz) ) {
         MessageBox(hwnd,
            "Memory allocation error",
            "Error", MB_OK | MB_ICONSTOP);
         hcDisconnect();
         exit(0);
         };
      if ( !hcGetRealArr(coordinates,
                  (double*)org_xyz,nAtmTot*3) ) {
         MessageBox(hwnd,
            "Error getting atomic coordinates",
            "Error", MB_OK | MB_ICONSTOP);
         hcDisconnect();
         exit(0);
         };
       }
      SetWindowText(hwndEdit[0],"12");
      // default step for rotation
      SetWindowText(hwndEdit[1],"30");
      // default number of rotations

      return 0;

   case  WM_COMMAND :
      hdc = GetDC (hwnd) ;
      SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
      switch (LOWORD (wParam)) {
      case 0 :// ROTATE
         EnableWindow(hwndButton[0], FALSE);
         GetWindowText(hwndEdit[0],szBuff,MAX_BUFF);
         dRot=atof(szBuff);
         GetWindowText(hwndEdit[1],szBuff,MAX_BUFF);
         nRot=atoi(szBuff);
```

```
      dRot = dRot*3.14159256/180.0;
      result=hcSetInt(cancel_menu,1);

      for (i=0;i<nRot;i++) {
         rAng=dRot*(i+1);
         rSin = sin(rAng);
         rCos = cos(rAng);
         for (ia=0;ia<nAtmTot;ia++) {
            new_xyz[ia].x = rCos * org_xyz[ia].x
               +  rSin * org_xyz[ia].y;
            new_xyz[ia].y =-rSin * org_xyz[ia].x
               +  rCos * org_xyz[ia].y;
            new_xyz[ia].z=org_xyz[ia].z;
            }
         result=hcSetRealArr(coordinates,
            (double*)new_xyz,nAtmTot*3);
         if (!hcGetInt(cancel_menu)) {
         result=hcSetRealArr(coordinates,
            (double*)org_xyz,nAtmTot*3);
         hcDisconnect();
         PostQuitMessage(0);
         break;
         }
      }
      result=hcSetInt(cancel_menu,0);
      EnableWindow(hwndButton[0], TRUE);

      break;
   default : ;
   }
            ReleaseDC (hwnd, hdc) ;
            ValidateRect (hwnd, &rect) ;
            break ;

case WM_DESTROY :

   result=hcDisconnect();
   PostQuitMessage (0) ;
   return 0 ;
       }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
  }
```

Executing this program (after invoking HyperChem) gives the following window:



Pushing the button will rotate the molecule by 12 degrees 30 times with the above parameters. In conjunction with this program the Cancel button functions appropriately. Pushing it will terminate the rotation and also terminate the rotate program.

If you are new to programming for Windows and NT, you should now have the basis for going on and beginning to build real Windows applications. However, development with the MFC, as described in the next chapter, will make your life much easier if you wish to develop serious graphical user interfaces.

Chapter 13

Development Using the MFC

Introduction

This chapter describes the development of C++ Windows programs using the Microsoft Foundation Classes (MFC). We describe very briefly how to develop Windows applications of this kind and then how to have them interface and exchange data with HyperChem programs. Examples of such programs are given.

Microsoft Development Tools

Once again, the appropriate development tool associated with this chapter is Microsoft Visual C++ which includes the Microsoft Foundation Classes. This set of C++ classes allow you to build a Windows graphical user interface in a very short period of time. The combination of the Integrated Development Environment (IDE), the C++ compiler, and the MFC classes, that are all part of Visual C++ 4.0 make for a powerful development tool. One can very quickly put together a Windows application, much faster than with the tools of the last chapter. It is still the case, however, that most of the commercial software, even that from Microsoft, does not yet use the MFC. As easy as it is, it does not provide quite the flexibility that making all your own lower-level API calls does, as in the SDK-style. In addition, it is still necessary to have a good appreciation of the Windows API, even if you are programming with the MFC. Indeed, the API calls are required for many things even within the higher level approach briefly described here. You should decide for yourself whether programming in C with direct API calls (Chapter 12) or programming in C++ with the MFC (this chapter) is for you.

Programming Assistance

As stated before, this cannot be a programming manual for Windows development. We will illustrate the basic ideas for building Windows MFC pro-

grams that interface to HyperChem and provide some example code but a serious approach to this subject requires you to obtain additional resources. The first, of course, is Microsoft Visual C++ 4.0 itself. The ideas discussed here can certainly be implemented with other compilers and with other development environments, but you will have to make some adaptation of the descriptions given here. We will not attempt to describe development with alternative tools.

The second requirement, if you are new to Windows development, is access to good documentation and tutorial material. One recent book on the subject from Microsoft is,

Programming Windows 95 with MFC

Jeff Proise

Microsoft Press, 1996

ISBN 1-55615-902-1

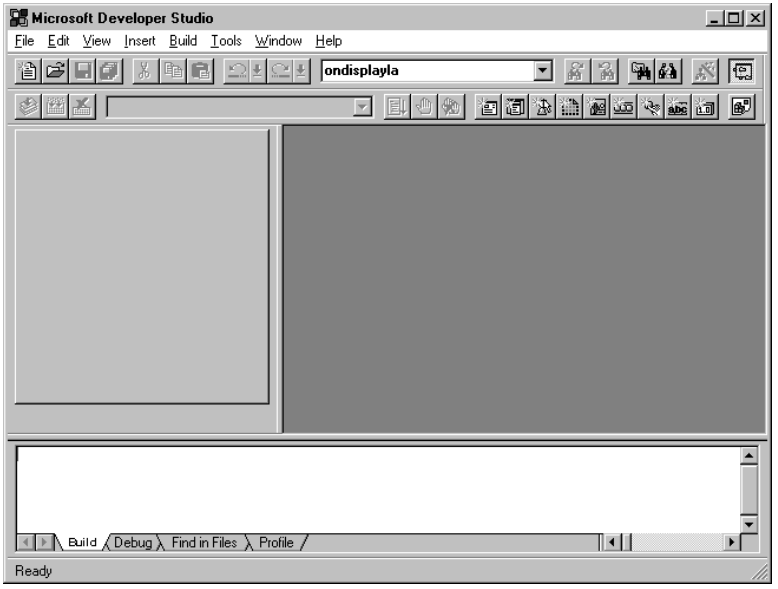
There are many other books that describe development using the MFC and Visual C++ 4.0 on the market, as well.

Language

The C++ language of this chapter may be new to you. If this is the case you may need programming books and tools that specifically address programming in this language. In many ways the object-oriented flavor of C++ makes it more different from C, than C even is from Fortran. An advantage of Visual C++ and its Wizards is that very little code will need to be written for development of the GUI and you will be able to develop significant applications just writing C code for the chemical computation part of your application. An investment in learning C++ programming is probably a well-rewarded investment.

A First Example

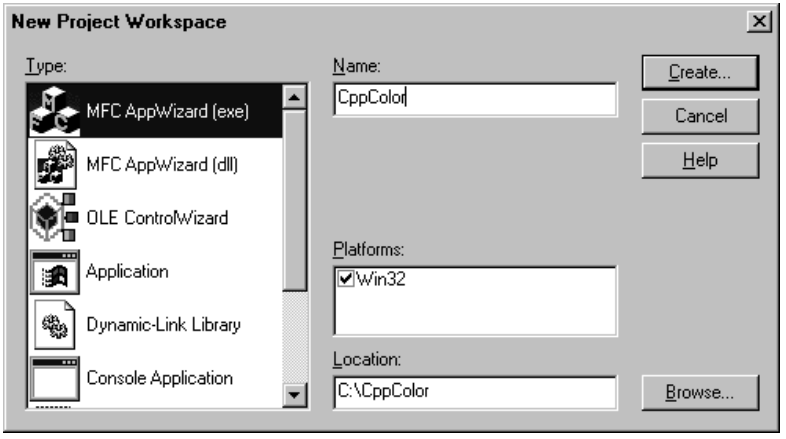
The first example of a C++ program that interfaces to HyperChem will be a simple one to illustrate the basic tools and concepts of Visual C++ 4.0. Bring up the Visual C++ program until it looks similar to the following:



Then,

1. Select <File/New...>

This will bring up a dialog box to select a Wizard to assist you in creating your application,



It is simplest to use the MFC AppWizard to create your application.

2. Type in a name for your application and hit <Create>.

You now are required to decide whether your application is a normal single document application with a menu bar, tool bar, etc., a corresponding document with a multiple document interface (MDI) or a simpler application that is essentially just a single dialog box. For this first application, choose the simpler approach.

3. Choose <Dialog based> and hit <Next>.

You are now asked to elaborate a little on the features of this dialog box.

4. Choose <About box> and <3D-controls> before hitting <Next>.

It is appropriate to choose a shared Dynamic Link Library for MFC and you can should comments that may assist you in understanding the files produced.

5. Choose <As a Shared DLL> and <Yes, Please> prior to hitting <Next>.

You can now complete the process of creating your first MFC application.

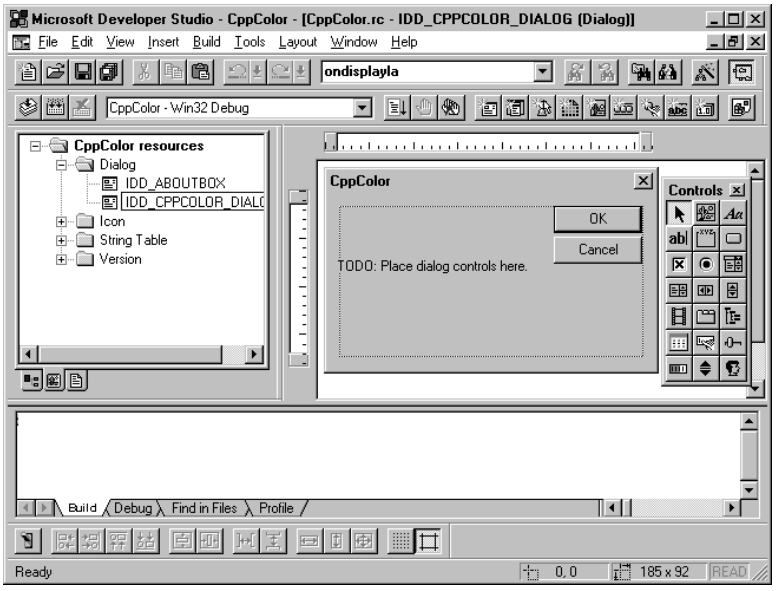
6. Choose <Finish> and <OK> to complete the AppWizard's work.

You are now left with an application that can be compiled and run but it will not do anything of significance yet or be able to interact with HyperChem.

Modifications

To create a custom application, we want to modify the widgets on the dialog box and create code for the modified ones. Specifically, we are once again going to create two buttons - one to change the HyperChem screen color to red and one to change it to green. To accomplish this, we first of all need to modify the existing resources and create two new resources that are the two new buttons.

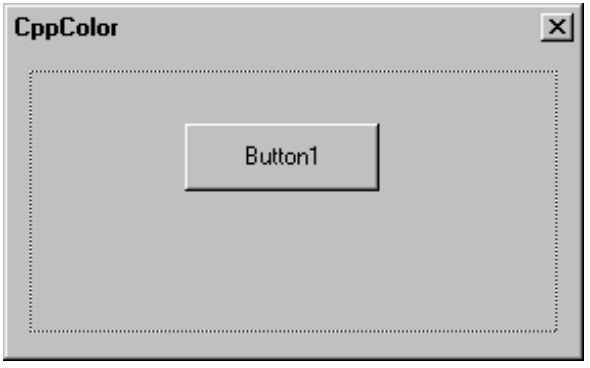
1. Click on the <ResourceView> tab at the bottom of the left window.
2. Double click on <IDD_CPPCOLOR_DIALOG> to place the current dialog box in the right window as shown below:



You will now want to delete the OK and Cancel buttons and the TODO label.

3. Click on <OK> to select the button and then <File/Delete> to delete it.
4. Repeat for the Cancel button.
5. Repeat for the TODO label.

You can now add whatever widgets you like to the dialog box. Select the Button control from the set of Controls and create a button on the dialog box by dragging with the mouse until you get the following:

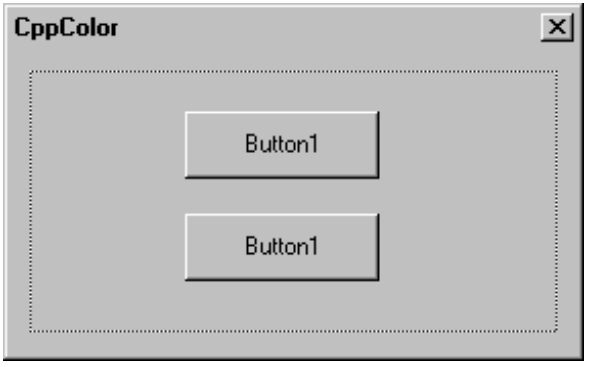


That is,

6. Create a Button on the dialog box.

To create a second button select the first one shown above, copy it to the clipboard and then paste it back,

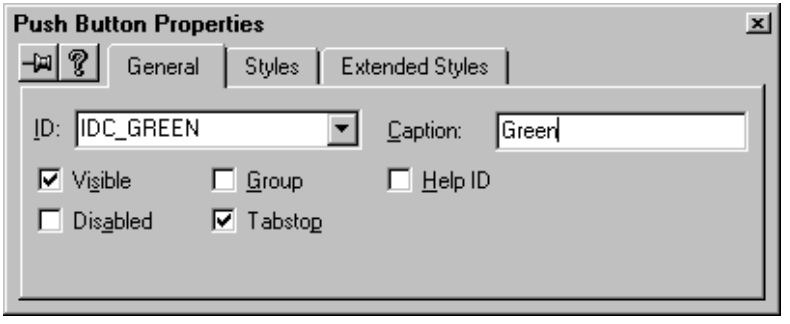
7. Copy and Paste the Button to create a second one.



You can now double click on each button in turn to modify its properties.

8. Double click on the first button to change its name to IDC_RED and it Caption to Red.

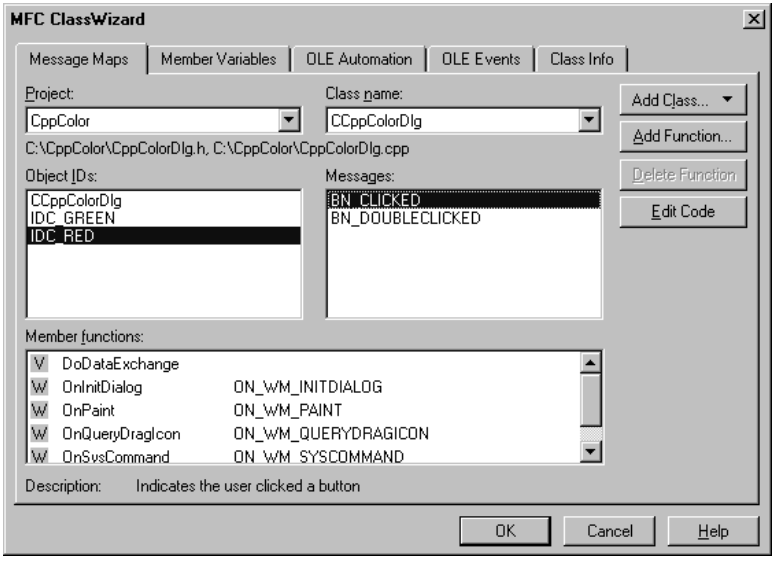
9. Repeat for the second Green button.



The buttons should now be labelled Red and Green. To continue, we need to invoke the Class Wizard to allow us to create specific code for these buttons.

10. Select the menu item <View/Class Wizard...>

You should see the following:



11. Select <IDC_RED> and <IDC_GREEN> in turn with the Message field as <BN_CLICKED> and hit <Add Function> to create functions OnRed and OnGreen.

Next you need to add code for these functions, so,

12. Hit <Edit Code> and type in the correct code for each button.

The appropriate code to add is:

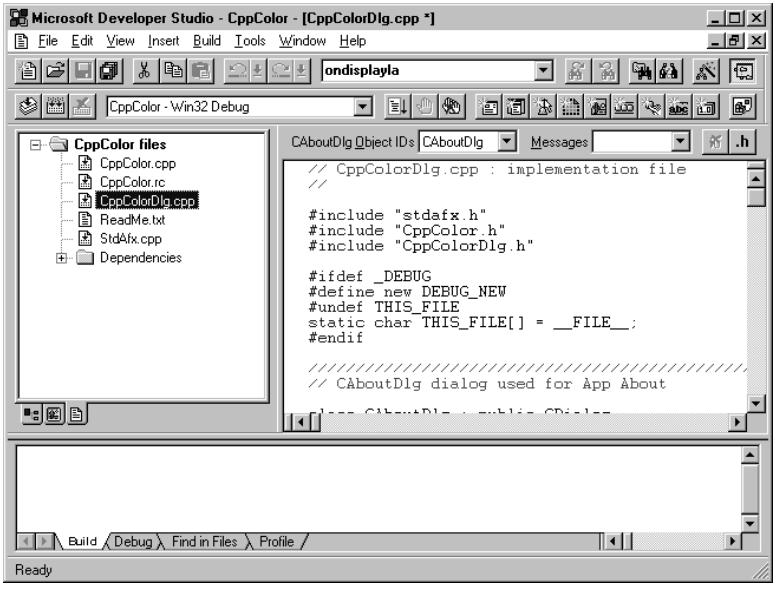
```
void CCppColorDlg::OnRed()
{
   // TODO: Add your control notification handler code here
   hcExecTxt ("window-color red");
}
```

13. Repeat steps 11 and 12 for the green button.

```
void CCppColorDlg::OnGreen()
{
   // TODO: Add your control notification handler code here
   hcExecTxt ("window-color green");
}
```

This completes the use of the AppWizard and Control Wizard for creating this application. There are still a couple of things that have to be done, however, before this code will compile and run correctly. The first task is to add our include files to the set of *includes*. The second task is to load the dynamic link library so that the function hcExec can be found. Finally we have to make a connection between this program and HyperChem.

The MFC application includes a number of files, all of which are machine generated. In this case CColor.cpp is really identical to a generic application and no changes to it are necessary. The CColorDlg.cpp file contains all the code associated with the dialog box and we need to modify this code. To look at code you select the <FileView> tab of the left window, find the file of interest and double click on it. This will give you a view of CColorDlg.cpp as below:



Included Files

At the beginning of the above file the following bold-faced code should be added to the include files already there:

```
// CppColorDlg.cpp : implementation file
//
#include "stdafx.h"
```

```
#include "CppColor.h"
#include "CppColorDlg.h"
#include "hc.h"
#include "hcload.c"
```

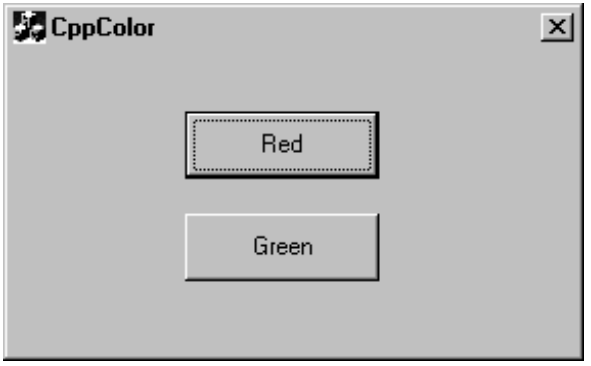
These are the include files that are needed to communicate with HyperChem. If you are making library calls, e.g. *hcExecBin*, you will need `hsv.h` also.

Dynamic Link Library and Connecting to HyperChem

The code that is needed to initialize the loading of the HAPI.DLL and establish the connection to HyperChem should be placed in the *OnInitDialog* routine inside `CColorDlg.cpp` after the appropriate place where the AppWizard tells you it should go,

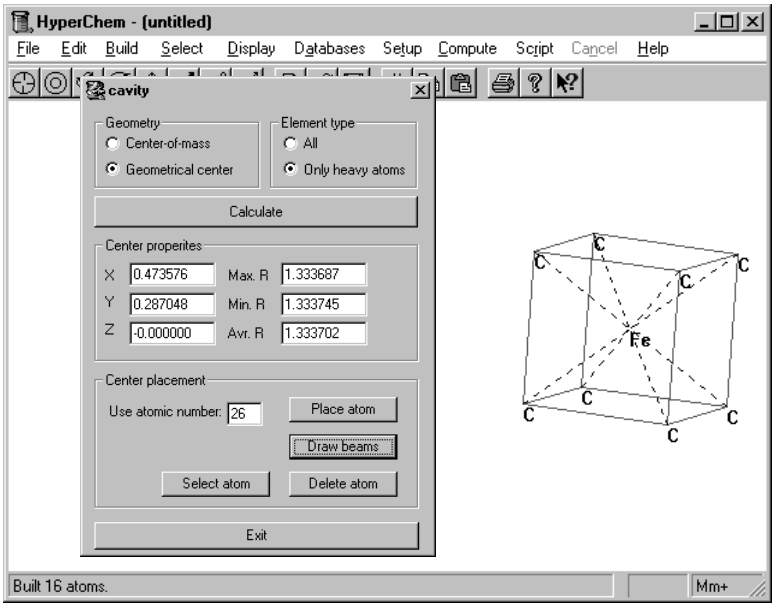
```
SetIcon(m_hIcon, TRUE);// Set big icon
SetIcon(m_hIcon, FALSE);// Set small icon

// TODO: Add extra initialization here

if (!LoadHAPI("hapi.dll"))
{
   MessageBox("Error loading HAPI.DLL", Error",
               MB_OK|MB_ICONSTOP);
   exit(0);
}
if (!hcConnect(NULL)))
   MessageBox("Error connecting to HyperChem", "Error",
               MB_OK|MB_ICONSTOP);
return TRUE;  // return TRUE  unless you set the focus to a control
```

This completes the first simple example. If everything went correctly, you should have created a Windows application that can talk to and control HyperChem. Remember that the code we have put into the new Windows application expects HyperChem to be there so that you need to start HyperChem before starting the new application. The CppColor application looks as follows:



Cavity

This next example, called Cavity, is considerably more sophisticated than the last example. It is still a simple Dialog application but it collects significant information from HyperChem - the current selections, the current coordinates, plus it collects the values of quantities in the dialog box before calculating the center of mass of the selection. The application can then place an atom of arbitrary atomic number at the center of mass of the “cavity” and draw special dotted bonds between the center of the cavity and the rest of the molecule. It is essentially identical to the last example except for the more sophisticated interactions with HyperChem and the inclusion of entries made in the dialog box. The code for this example is on the HyperChem CD-ROM. The dialog box is shown below,



This application is too long to discuss all the code here. If you are interested, however, you should be able to follow every aspect of this relatively simple calculation using the full source code. All the calls in this program are text-based calls with no binary call that need hsv.h. We will discuss certain portions of the code as being instructive of how one generally interacts with HyperChem. The code, *OnCalc*, associated with pushing the Calculate button is where the coordinates are read. The first thing that is necessary, before reading the coordinates, is to characterize the atoms and molecules of the system in HyperChem,

```
// HyperChem won't send tags with OMSGS
hcExecTxt("query-response-has-tag = false");
// Gather information about molecular system
// start with number of molecules
resp=hcQueryTxt("molecule-count");
iMol=atoi(resp);hcFree(resp);
// allocate memory for iaAtomCount array
iaAtomCount=(int*)calloc(iMol,sizeof(int));
// next, get count of atom in each molecule
resp=hcQueryTxt("atom-count");
// parse string returned by CDK into integer array
```

```
pstr=resp;
i=0;iAtomTot=0;
while (ptok=strtok(pstr,_DELIMITERS)) {
   if (pstr) pstr=NULL;
      iAtomTot += iaAtomCount[i++]=atoi(ptok);

}
hcFree(resp);
```

As with many interfaces to HyperChem, the first thing to do is to eliminate the tag which comes, by default, with a returning message. Next, we get the number of molecules, *molecule-count*, which comes in as a string and gets converted to an integer. The memory allocation for the string occurs automatically within hcQueryTxt and this memory ***must be freed*** by you when the string is no longer needed. Forgetting to free memory is a common C programming error.

The HSV, atom-count, is a vector. Each element could have been read in turn but all the elements can also be read at once - the resulting string must then be parsed, however. The vector , iaAtomCount, is used to hold the result of this parsing.

The next step is to sort out which of the atoms and molecules are selected since we are only going to read the selected atoms coordinates. This result in vectors iaSelectedAtom and iaSelectedMol being allocated and filled in, plus a variable, iSelected, that represents the total number of atoms selected. The code for this can be inspected in the source files if you wish. We skip it for brevity. The next step is to read the coordinates for the selected atoms. The code for this is,

```
// get coordinates of atoms
faX=(float*)calloc(iSelected,sizeof(float));
faY=(float*)calloc(iSelected,sizeof(float));
faZ=(float*)calloc(iSelected,sizeof(float));
for (is=0;is<iSelected;is++) {
   wsprintf(buff,"coordinates(%d,%d)",
         iaSelectedAtom[is],iaSelectedMol[is]);
   resp=hcQueryTxt(buff);
   pstr=resp;
   i=0;in=0;
   while (ptok=strtok(pstr,_DELIMITERS)) {
      if (pstr) pstr=NULL;
      switch (in)  {
```

```
            case 0: x=(float)atof(ptok); in++; break;
            case 1: y=(float)atof(ptok); in++; break;
            case 2: z=(float)atof(ptok); in=0; break;
         }

      }
      hcFree(resp);
      faX[is] = x;
      faY[is] = y;
      faZ[is] = z;
}
```

The loop over selected atoms makes a request for the coordinates of that atom, as an element of the coordinate array. This element is a string of three values (x, y, and x coordinates) and has to be parsed.

The code for the remaining portions of this problem is available for your inspection, if you are interested.

Chapter 14
Console C and Fortran Applications

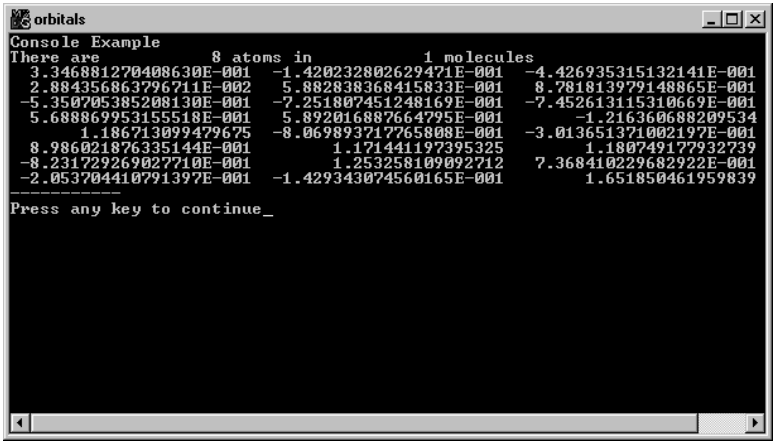
Introduction

This chapter describes how to develop “conventional” character-oriented or console applications in C or Fortran that can be interfaced with HyperChem.

It is relatively common in computational or theoretical chemistry to develop applications in Fortran (or as is becoming more common, in C) that have no graphical user interface but operate with input files creating output files. For example, many programs have been developed to calculate a molecular wave function. Such a program in its simplest form might need as input the Cartesian coordinates of a molecule and give as output the energies and coefficients describing the molecular orbitals. It is a significant effort to build the additional graphics program that would allow the molecule to be just sketched and the output to be presented as a 3D rendering of the molecular orbitals. The CDK, however allows the developer of such a molecular orbital program to use HyperChem to sketch the molecule and to render the molecular orbitals. HyperChem can act as a front end GUI and a back end visualizer to this character-oriented program by simply having it call on HyperChem whenever it needs graphical services.

Console Applications

Microsoft refers to programs that have no graphical user interface as console applications. An example output from one of these console applications is shown below.



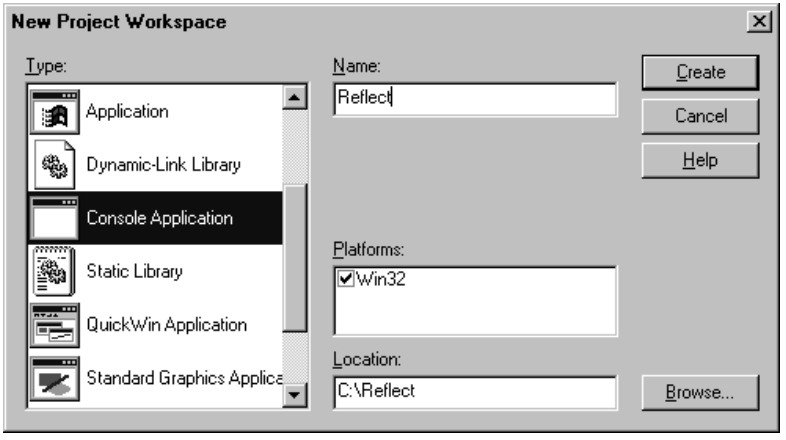
These applications look a lot like many UNIX applications or applications developed in DOS. Until recently it was very difficult to develop such console programs within the Windows environment, i.e. without a window and a graphical user interface. Now with Windows 95 or NT and Visual C++ 4.0 and Fortran Power Station it is relatively easy to port traditional non-graphical scientific applications to the Windows environment. With the Chemists Developer Kit, it is also easy to interface these to HyperChem.

C or Fortran

Until recently, the Microsoft Windows environment and development tools have essentially been usable only with C (or C++). With the recent introduction of Fortran PowerStation, however, it has become easy to develop Windows applications in Fortran. For console applications, the tools for C and Fortran are essentially on the same basis and the choice can be based upon which language is your personal choice. For more graphical programs, Fortran (which requires calls to the Windows API - developed for C) is still more awkward to use than C. It is still possible, however, using tools that Microsoft has provided with Fortran PowerStation. In this chapter and in this manual we will not concern ourselves with development of graphical user interfaces in Fortran. Rather, we restrict our Fortran discussions to the console applications of this chapter. For such console applications, C and Fortran are interchangeable as to their ease of use within Windows.

The Integrated Development Environment

The Visual C++ 4.0 Integrated Development Environment is where Fortran PowerStation programs are developed and this gives an identical development environment for both C and Fortran. Thus a console program is created, as before, by selecting <File/New>, choosing to create a new *Project Workspace* and then selecting a Console Application,



For console applications, one has to *Insert* C (*.c), C++ (*.cpp), or Fortran (*.f) files into the project as needed since none are generated automatically as with the AppWizard.

C Program

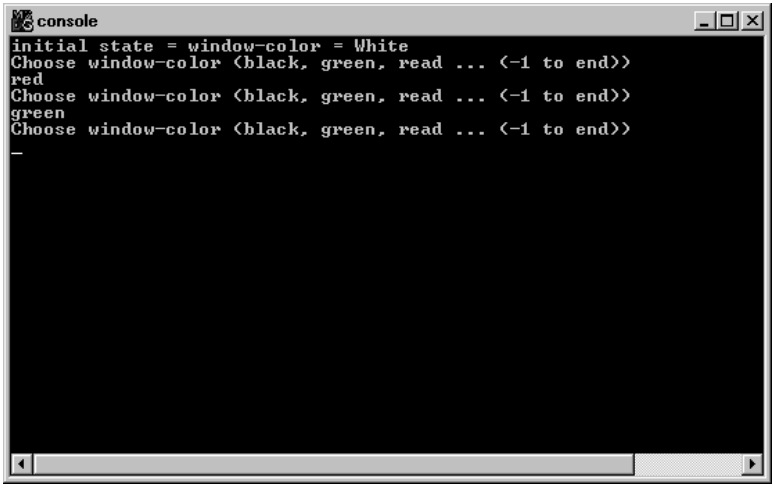
Our first example of a console program is a C program. The program looks like any C program beginning with main(), etc. The code for it is as follows:

```
/*
 * Console - UNIX like C program that talks to HyperChem *
 */
#include <stdio.h>
#include <windows.h>
#include "hc.h"
#include "hcload.c"
int main(int iArg, char **pArg)
{
char cmd_line[100],initColor[100],color[100],buffer[100];
```

```
char *response;
int result;
/* loading HAPI.DLL   */
   if (!LoadHAPI("hapi.dll")) {
      printf("Error loading HAPI.DLL !\n");
      exit(0);
   };
/* connecting to HyperChem  */
   if (iArg!=2) {
      strcpy(cmd_line,"");
   } else {
      strcpy(cmd_line,pArg[1]);
   };
   if (!hcConnect(cmd_line)) {
      printf("Error while connecting with HyperChem !\n");
      exit(0);
   }

/* obtain initial window-color from HyperChem */
response=hcQueryTxt("window-color");
printf("initial state = %s\n",response);
strcpy(initColor,response);
hcFree(response);

for(;;) {
   printf("Choose window-color (black, green, red ... (-1 to end))\n");
   scanf("%s",color);
   if (atoi(color) == -1) {
      result=hcExecTxt(initColor);
      result=hcDisconnect();
      return 0 ;
   } else {
      sprintf(buffer,"window-color %s",color);
      result=hcExecTxt(buffer);
   }

}
return 0;
```

This code looks like any simple C program with the exception of calls to hcConnect, hcQueryTxt, hcExecTxt, and hcDisconnect. Any command line arguments are passed through to HyperChem. The header files, hc.h and hcload.c are required as is the dynamic loading of the library of the HyperChem Application Programming Interface, HAPI.DLL. Subsequent to con-

necting to HyperChem, the program can just read and write HSV's to HyperChem or control HyperChem via direct commands. The program requires HyperChem to be running prior to its own invocation. The result of running this program is the following Console window.



The corresponding HyperChem window with its changing background colors is not shown. The console window contains any text that a normal C program outputs via "standard output" using printf, etc. It can be iconized, if desired, so that it is effectively running in the background while you interact with HyperChem. It behaves very much like HyperNewton, HyperNDO, etc., the standard back ends that come with HyperChem and perform compute intensive computations.

Fortran Programs

We will now illustrate, in somewhat more detail, examples of Fortran programs that behave like the above. It is believed that HyperChem and the CDK provide an ideal opportunity to interface a wide variety of Fortran programs, such as could be available from the Quantum Chemistry Program Exchange (QCPE) at Indiana University. Many programs have been generated over many years, by many chemists, and these could very quickly have a "slick" graphical user interface.

Reflect

Our first example is a program that simply collects the coordinates of all the atoms from HyperChem, performs some simple transformation of these coordinates, and sends them back to HyperChem for display. For simplicity we will use a reflection in the XY plane through the origin to illustrate the general process. That is, our program will replace all Z coordinates by -Z. If your molecule has chiral centers and their chirality label is displayed, you will see the conversions R->S and S->R from running this program.

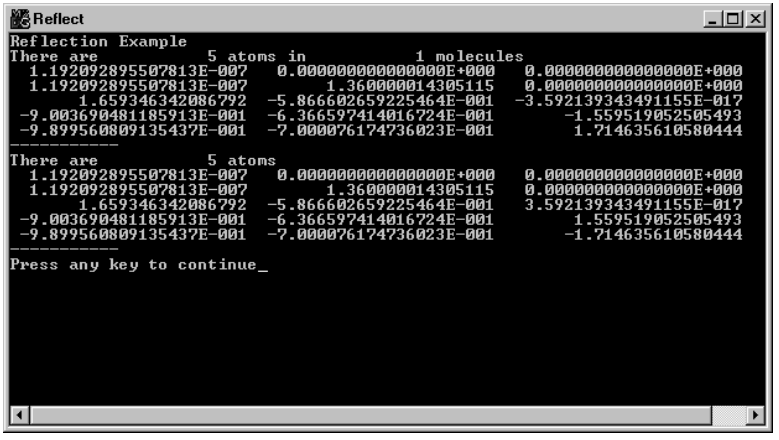
This example, while trivial, has some of the characteristics of a "real" program that would perform significant computation on an "initial" structure leading to a "new" structure. Molecular dynamics programs or a structure optimization programs have this flavor. The whole program is,

```
program Reflect
c-----------------------------------------------------------------------
c include header files - HAPI definitions and declarations
   include 'hc.fi'
   include 'hsv.fi'
   parameter (nDimensions=3)
   parameter (nMaxAtoms=2000)

   character*60 cmd_line
   logical result
   integer status

   integer nAtoms
   double precision XYZ(nDimensions, nMaxAtoms)

c
c Connect to HyperChem using current command line
c
   call getarg(1,cmd_line,status)
   result=hfConnect(cmd_line)
   if (.NOT. result) stop

c we do not want to have names in front of HSVs !
c query-response-has-tag=false does this for us
   result=hfExecTxt("query-response-has-tag=false")

   write(*,*)'Reflection Example'

c call subroutines to read in initial atom coordinates
```

```
c and write final atom coordinates
   call GetCoords(XYZ, nDimensions, nMaxAtoms, nAtoms)
   call PutCoords(XYZ, nDimensions, nMaxAtoms, nAtoms)

   end

   subroutine GetCoords(coords,nDim,nMaxAt,nAt)
c-------------------------------------------------
c gets xyz coordinetes of all atoms in the system
c and stores them in 'coords' array
c-------------------------------------------------
c include HAPI definitions and declarations
   include 'hc.fi'
   include 'hsv.fi'

   double precision coords(nDim,nMaxAt)

c get number of molecules from HyperChem (integer value)
c using hfGetInt "binary" function

   nMol=hfGetInt(molecule_count)

c get total number of atoms by scanning all molecules
   nAt=0
   do 1 i=1,nMol

c get number of atoms in the molecule
c hfGetIntVecElm is "binary" function ("atom_count" is Integer Vector)
   iatoms=hfGetIntVecElm(atom_count,i)
   nAt=nAt+iatoms
1  continue
   write(*,*)'There are ',nAt,' atoms in ',nMol,' molecules'


c get xyz coordinates of all atoms and place them into 'coords' array
c using hfGetRealArr "binary" function ("coordinates" is Real Array)
   lres=hfGetRealArr(coordinates,coords,nDim*nMaxAt)

   do 111 i=1,nAt
   write(*,*)(coords(k,i),k=1,3)
111continue
   write(*,*)'-----------'
   return
```

```
   end

   subroutine PutCoords(coords,nDim,nMaxAt,nAt)
c-------------------------------------------------
c takes the coords array and send it back to HyperChem
c as the cartesian coordinates of all atoms
c-------------------------------------------------
c include HAPI definitions and declarations
     include 'hc.fi'
     include 'hsv.fi'

     double precision coords(nDim,nMaxAt)

     write(*,*)'There are ',nAt,' atoms'
     do 111 i=1,nAt
     coords(3,i) = -coords(3,i)

     write(*,*)(coords(k,i),k=1,3)
111  continue
     write(*,*)'-----------'

c get xyz coordinates of all atoms and place them into 'coords' array
c using hfGetRealArr "binary" function ("coordinates" is Real Array)
   lres=hfSetRealArr(coordinates,coords,nDim*nAt)
     return
     end
```

The console output is,



This example illustrates the use of both binary calls such as *hfSetRealArr* that require the include file `hsv.fi` as well as text calls such as *hfExecTxt* that do not. The binary calls require the include file `hsv.fi` to map integers such as “atom_count” to the appropriate HyperChem variable. Note that in binary calls all “hyphens” are replaced by “underscores”. Thus the HSV, atom-count, maps to the integer atom_count through `hsv.fi`. Each of the HAPI calls is described in Appendix C.

A single call in the code above is all that is necessary to read or write the atom coordinates once the total number of atoms is know. HyperChem combines atom numbers within a molecule with molecule numbers to obtain a unique atom number. The code first of all has to query for the number of molecules and then for the number of atoms in each molecule to obtain the total number of atoms. Beyond that computation the rest of the above program is very straight forward.

MiniGauss Orbitals

The next example is one which contains the elements of a number of potentially very significant uses for the CDK. It provides molecule creation and visualization for an *ab initio* wave function package. The *ab initio* package is based on a demonstration Fortran code that is an appendix to the book,

Modern Quantum Chemistry

Attila Szabo and Neil S. Ostlund

Dover Publications, Inc.

New York, 1996

This program, only a couple of pages long, contains all the code for an *ab initio* STO-1G, STO-2G, or STO-3G calculation on 2-electron diatomics like H_2, HeH^+, He_2^{++}, etc. It has proved a useful educational tool for a number of young (and not-so-young) theoreticians. What we do here is offer a “front end” to this program to illustrate how HyperChem can provide molecular coordinates plus the visualization of the 3D shape of the calculated orbitals and charge density.

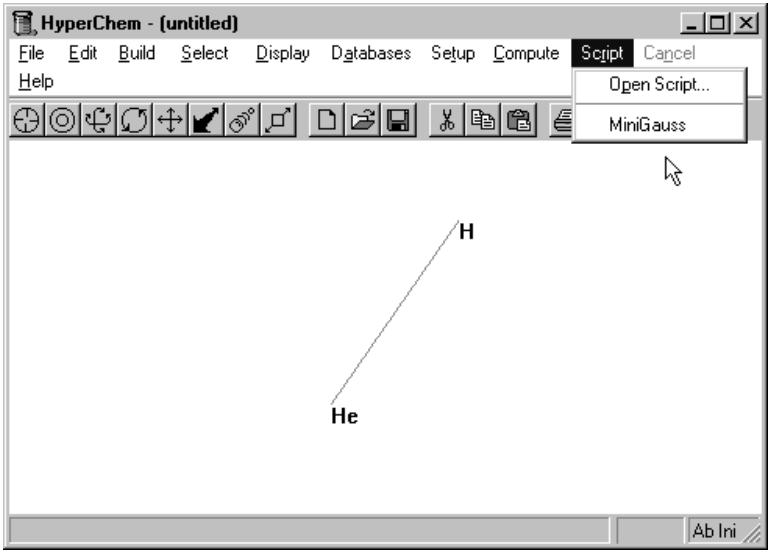
Outline

The basic idea of this example is to mimic the actions of HyperGauss, a full fledged ab initio package that comes with HyperChem. We will operate the

MiniGauss back end by having a menu item and scripts to conveniently run the Fortran program. All the scripts for this example, plus the Fortran code, are on the HyperChem CD-ROM associated with the *Orbitals* directory. While we could easily use the idea of custom menus to run this program we will simply place an appropriate menu item in the <Script> menu. To do this, we first of all run a script, *orbitals.scr*, to set up a convenient way of running these calculations. This script is,

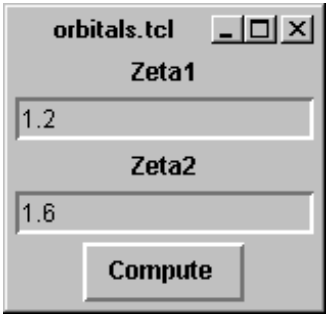
```
script-menu-caption(1) =  "MiniGauss"

script-menu-enabled(1) = true

script-menu-command(1) = "read-tcl-script orbitals.tcl"
```

This sets up a new menu item, *MiniGauss*, that when invoked executes a Tcl script called `orbitals.tcl`. The `orbitals.scr` script could be placed into `chem.scr` so that it is always executed and the MiniGauss menu always appears - if you have developed an application that you would like to be more or less permanently installed. The HyperChem menu item looks as follows:



A New GUI Element

The Tcl/Tk script, `orbitals.tcl` is going to give us an additional GUI element (dialog box) that makes for interactive use of MiniGauss. This dialog box, invoked by selecting the MiniGauss menu item above, is,



It collects the two Slater exponents for the minimal basis calculation. For the other input to the calculation, i.e. the N of STO-NG, we are going to use the standard facility of HyperChem to define a basis set. Thus, it is only the above orbital exponents, plus of course the molecule, that MiniGauss needs. The Tcl/Tk script, `orbitals.tcl`, is,

```
label .l1 -text "Zeta1"
entry .en1 -width 20 -textvariable inzeta1
label .l2 -text "Zeta2"
entry .en2 -width 20 -textvariable inzeta2
button .b -text "Compute" -command {
hcExec "declare-string zeta1"
hcExec "declare-string zeta2"
hcExec "zeta1 $inzeta1"
hcExec "zeta2 $inzeta2"
hcExec "execute-client mini.exe"
Exit
}
pack .l1 .en1 .l2 .en2 .b
```

This Tcl script is basically just Tk code to create two labels, two text entry boxes, and a button to dismiss the dialog box and start the MiniGauss calculation.

The two text entry boxes are used to input the two orbital exponents, zeta1 and zeta2, as strings inzeta1 and inzeta2. The button calls HyperChem to create two new string variables (new HSV's) called appropriately zeta1 and zeta2. These new HSV's are then assigned the strings collected from the entry boxes. This is the first time we have seen the ability to create new HSV's and as you can see here it is very useful to provide a repository for any new values entered from a new GUI until they can be passed to their ultimate destination, which in this case will be the MiniGauss program.

The Main Program

The last two things done by pushing the Compute button are:

```
hcExec "execute-client orbitals.exe"
Exit
```

The Exit, with a capital E, exits the Tcl script but not HyperChem [an exit with a small e would exit both the Tcl script and HyperChem!]. In concert with this, the MiniGauss program is invoked which is our main Fortran program and is here called `orbitals.f` (and `orbitals.exe`).

```
Program Orbitals
   implicit double precision(a-h,o-z)
c -----------------------------------------------------
c include header files - HAPI definitions and declarations
   include 'hc.fi'
   include 'hsv.fi'
   parameter (nDimension=3)
   parameter (nMaxAtoms=2000)

   character*60 cmd_line
   logical result
   integer status

   integer nAtoms
   double precision XYZ(nDimension, nMaxAtoms)

c
c Connect to HyperChem using current command line
c
   call getarg(1,cmd_line,status)
```

```
   result=hfConnect(cmd_line)
   if (.NOT. result) stop

c we do not want to have names in front of HSVs !
c query-response-has-tag=false does this for us
   result=hfExecTxt("query-response-has-tag=false")

   write(*,*)'Orbitals Example'

c call subroutine to read in initial atom coordinates
c
   call DoCalc(XYZ, nDimension, nMaxAtoms, nAtoms)
   call SendResults

   end
```

The main program simply connects to HyperChem and makes sure that you don't get the tags along with the values when HSV's are queried. It then calls a routine (DoCalc) that is principally concerned with performing the calculation and one which sends back the results to be displayed graphically (SendResults).

Get Molecule

The Subroutine DoCalc is primarily concerned with getting the coordinates of all the atoms (nAtoms) to be input to the minimal basis set *ab initio* calculation. This subroutine is,

```
   subroutine DoCalc(coords,nDim,nMaxAt,nAt)
   implicit double precision(a-h,o-z)
c-------------------------------------------------
c gets xyz coordinates of all atoms in the system
c and stores them in 'coords' array
c-------------------------------------------------
c include HAPI definitions and declarations
   include 'hc.fi'
   include 'hsv.fi'

   character *100 buffer
   double precision coords(nDim,nMaxAt)

c get number of molecules from HyperChem (integer value)
c using hfGetInt "binary" function
```

```
   nMol=hfGetInt(molecule_count)
c we can work only with one molecule
   if (nMol .gt. 1) then
   write(*,*)'This demo assumes that you have only ONE molecule'
c disconnect from HyperChem
   result=hfDisconnect()
   stop
   endif

c get total number of atoms by scanning all molecules
   nAt=0
   do 1 i=1,nMol

c get number of atoms in the molecule
c using hfGetIntVecElm "binary" function ("atom_count" is Integer
Vector)
      iatm=hfGetIntVecElm(atom_count,i)
      nAt=nAt+iatm
1  continue
c we can only work with a diatomic
   if (nAt .ne. 2) then
   write(*,*)'This demo assumes that you have TWO atoms'
c disconnect from HyperChem
   result=hfDisconnect()
   stop
   endif
c find out atomic numbers
   nAtNum1=hfGetIntArrElm(atomic_number, 1, 1)
   nAtNum2=hfGetIntArrElm(atomic_number, 2, 1)
   if (nAtNum1 .gt. 2 .or. nAtNum2 .gt. 2) then
   write(*,*)'This demo assumes that you have H or He atoms'
c disconnect from HyperChem
   result=hfDisconnect()
   stop
   endif
   za = dfloat(nAtNum1)
   zb = dfloat(nAtNum2)
   nCharge=hfGetInt(quantum_total_charge)
   nElectrons = nAtNum1 + NAtNum2 - nCharge
   if (nElectrons .ne. 2) then
   write(*,*)'This demo assumes that you have 2 electrons'
c disconnect from HyperChem
```

```
   result=hfDisconnect()
   stop
   endif
c get the basis set
   result = hfQueryTxt("atom-basisset(1,1)", buffer)
   if (buffer(1:6).eq.'STO-3G') then
    n=3
   else if (buffer(1:6) .eq. 'STO-2G') then
    n=2
   else if (buffer(1:6) .eq. 'STO-1G') then
    n=1
   else
    n=3
   end if

c get the zeta
   result = hfQueryTxt("zeta1", buffer)
   read (buffer,*) zeta1
   result = hfQueryTxt("zeta2", buffer)
   read (buffer,*) zeta2

c get xyz coordinates of all atoms and place them into 'coords' array
c using hfGetRealArr "binary" function ("coordinates" is Real Array)
   lres=hfGetRealArr(coordinates,coords,nDim*nMaxAt)

   dx = coords(1,1)-coords(1,2)
   dy = coords(2,1)-coords(2,2)
   dz = coords(3,1)-coords(3,2)
   r = dsqrt( dx*dx + dy*dy + dz*dz) /0.52918

   call hfcalc(n,r,zeta1,zeta2,za,zb)
   return
   end
```

The above code contains a number of text and binary calls to HyperChem. A single call (hfGetRealArr) gets all the coordinates once we know the number of atoms. The two orbital exponents that we got from the Tk dialog box and stored in HyperChem are retrieved from HyperChem along with other information that HyperChem has about the calculation that is about to be performed. One aspect of this information is the basis set, as set by the user in HyperChem. If HyperChem says it is one of the STO-NG basis sets then that information is used, otherwise an STO-3G calculation is performed. The remaining code near the beginning is associated with getting basic information about the molecule and making sure it is simple enough for MiniGauss.

Wave function Calculation

The call to the routine *hfcalc* is set up to be identical (apart from iop which is a printing option) to that in Szabo and Ostlund so that you might substitute their few pages of code if you so desire. A drastically simpler version of hfcalc used here is,

```
subroutine hfcalc(n,r,zeta1,zeta2,za,zb)
   implicit double precision(a-h,o-z)
   common/matrix/s(2,2),x(2,2),xt(2,2),h(2,2),f(2,2),g(2,2),c(2,2),
   $ fprime(2,2),cprime(2,2),p(2,2),oldp(2,2),tt(2,2,2,2),e(2,2)
   c(1,1) = .707
   c(2,1) = .707
   c(1,2) = .707
   c(2,2) = -0.707
   e(1,1) = -5.0
   e(2,2) = 5.0
   return
   end
```

This just sets the eigenvalues to arbitrarily be -5.0 and +5.0 and the eigenvectors to be the standard in phase and out of phase orbitals for HOMO and LUMO. The assumption here is made that the overlap is small so that coefficients are just 1/sqrt(2).

As discussed a better hfcalc could easily be programmed. The one shown here is perfectly satisfactory for illustrating the use of the CDK, however.

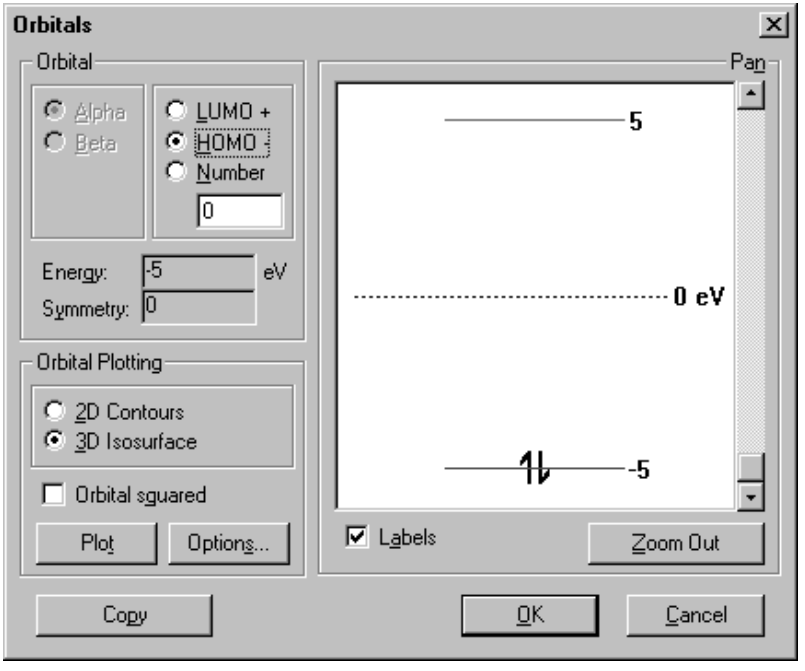
Displaying Orbitals and ...

The routine that sends back the orbitals for display is,

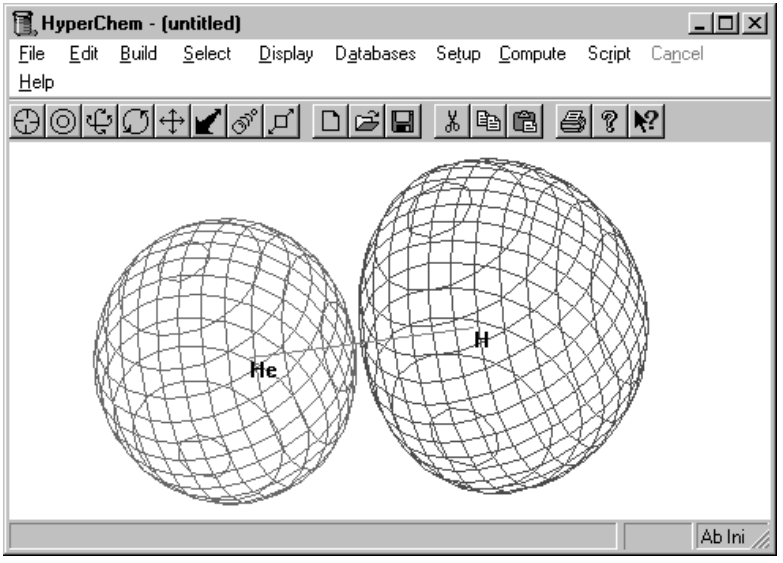
```
subroutine SendResults
   implicit double precision(a-h,o-z)
   common/matrix/s(2,2),x(2,2),xt(2,2),h(2,2),f(2,2),g(2,2),c(2,2),
     $ fprime(2,2),cprime(2,2),p(2,2),oldp(2,2),tt(2,2,2,2),e(2,2)

c include HAPI definitions and declarations
   include 'hc.fi'
   include 'hsv.fi'

   character *80 buff

   result = hfExecTxt('orbital-count=2')
   write (unit = buff, fmt = '(2f8.4)') e(1,1), e(2,2)
```

```
result = hfExecTxt('scf-orbital-energy='//buff)
write (unit = buff, fmt = '(2f8.4)') c(1,1), c(2,1)
result = hfExecTxt('alpha-scf-eigenvector(1)='//buff)
write (unit = buff, fmt = '(2f8.4)') c(1,2), c(2,2)
result = hfExecTxt('alpha-scf-eigenvector(2)='//buff)
write result = hfExecTxt('alpha-orbital-occupancy= 2 0')
return
end
```

There are a number of ways that the results might be returned. One way is to make binary HAPI calls such as to *hcSetRealVec* to return vectors of coefficients. We have chosen here to illustrate the return of text strings to HyperChem. The first thing that is done is to set the total count of the number of molecular orbitals which in the minimal basis illustrated here is 2. Next the orbital energies are returned and then the eigenvectors. Since this is only a closed-shell calculation, only the alpha (RHF) coefficients need be filled in. Finally the occupancy is set with 2 electrons in the first orbital (HOMO) and 0 electrons in the LUMO. The result of selecting the MiniGauss menu item with HeH^+ on the screen is,



The orbitals or the charge density or the electrostatic potential can be plotted The orbitals are somewhat artificial here but asking for the LUMO orbital gives a plot like the following:



Diffusion Limited Aggregation

The final example to be mentioned is one of diffusion limited aggregation and is contained in the dla directory from the HyperChem CD-ROM. This is presented for those who wish to follow it as an example that has a Visual Basic program, setup_dla.exe, to create a number of new variables in HyperChem, the getting from HyperChem of the initial molecular system, the creation of new atoms and structures in the HyperChem workspace, and a more elaborated dialog box than we have used so far.

Further Examples

Further examples of using the CDK that might be of interest to you can possibly be found in conjunction with your specific CD-ROM installation of HyperChem or possibly on the Hypercube WWW site, http://www.hyper.com

Appendix A
Classification of Hcl Commands

The Classes

In this appendix we list all the HSV's and Hcl direct commands according to a set of classes that will hopefully assist you in finding the appropriate script command for your task. In the Reference Manual these script commands are classified differently - according to the related HyperChem menu. In Chapter 7 all HSV's and direct Hcl commands are listed in alphabetic order. These three listings complement each other and should assist you in getting familiar with the extensive set of script commands. The listings in the Reference Manual have the most explicit description of each of the following script commands and should be referred to if the use of the command listed here or in Chapter 6 is confusing to you.

The classes which we use to help classify all the script commands are:

- General Operations
- Cursors
- Selections
- File Operations
- Scripts
- Info
- Errors
- Logging
- Auxiliary
- Viewing
- Rendering
- Coloring and Labeling

- Images
- Model Building
- Stereochemistry
- Atom Properties
- Molecular Properties
- Backends
- Molecular Mechanics Calculations
- Amino Acids and Nucleic Acids
- Molecular Dynamics and Monte Carlo
- Optimization
- General Quantum Mechanics
- Semi-empirical Calculations
- Ab Initio Calculations
- Configuration Interaction
- Infrared Spectra
- UV Spectra
- Plotting

General Operations

The following script commands involve the general operation of HyperChem including things that don't easily fit into other categories. Thus, in this category we have the script commands for a single-point calculation and for solvation using the periodic box. Other commands have to do with default or custom menus, printing, and the operation of the Cancel button.

Single Point

calculation-method: Variable, Read/Write, Type: enum.

do-single-point: Command, Arg list: (void).

total-energy: Variable, Readonly, Type: float in range (-1e+010 .. 1e+010).

Solvation

periodic-boundaries: Variable, Read/Write, Type: boolean.

periodic-box-size: Variable, Readonly, Type: (unknown).

solvate-system: Command, Arg list: (void).

solvate-system-in-this-box: Command, Arg list: float, float, float.

Customization

hide-toolbar: Variable, Read/Write, Type: boolean.

load-default-menu: Command, Arg list: (void).

load-user-menu: Command, Arg list: string.

switch-to-user-menu: Command, Arg list: (void).

custom-title: Variable, Read/Write, Type: string.

factory-settings: Command, Arg list: (void).

Printing

print: Command, Arg list: (void).

printer-background-white: Variable, Read/Write, Type: boolean.

Other

cancel-menu: Variable, Read/Write, Type: boolean.

do-vibrational-analysis: Command, Arg list: (void).

help: Command, Arg list: string.

hide-messages: Variable, Read/Write, Type: boolean.

Cursors

The following script commands have to do with the operation of the various cursors apart from the drawing and selection cursor. The remaining cursors are associated with rotation, translation, zooming, and clipping operations. These have parameters that describe, for example, the unit of rotation when the rotation cursor is dragged in the work space (x-y-rotation-cursor). alternatively, there is a unit or rotation when the appropriate keyboard equivalent is

used (x-y-rotation-icon-step). These parameters can be set via the <File/Preferences...> dialog box or via the following script commands.

Mouse Mode

mouse-mode: Variable, Read/Write, Type: enum

Clipping

clip-cursor: Variable, Read/Write, Type: float in range (0 .. 1000).

clip-icon-step: Variable, Read/Write, Type: float in range (0 .. 1000).

back-clip: Variable, Read/Write, Type: float

front-clip: Variable, Read/Write, Type: float.

Rotation

x-y-rotation-cursor: Variable, R/W, Type: float angle in range (0 .. 3600).

x-y-rotation-icon-step: Variable, R/W, Type: float angle in range (0 .. 3600).

z-rotation-cursor: Variable, R/W, Type: float angle in range (0 .. 3600).

z-rotation-icon-step: Variable, R/W, Type: float angle in range (0 .. 3600).

Translation

x-y-translation-icon-step: Variable, R/W, Type: float in range (0 .. 1000).

z-translation-cursor: Variable, R/W, Type: float in range (0 .. 1000).

z-translation-icon-step: Variable, R/W, Type: float in range (0 .. 1000).

Zoom

zoom-cursor: Variable, Read/Write, Type: float in range (1 .. 1000).

zoom-icon-step: Variable, Read/Write, Type: float in range (1 .. 1000).

Selections

The following script commands have to do with making selections, a fundamental operation. HyperChem generally operates on a selection rather than on the whole molecular system and these selection scripts are often used in conjunction with other operations.

Select Options

multiple-selections: Variable, Read/Write, Type: boolean.

select-sphere: Variable, Read/Write, Type: boolean.

selection-target: Variable, Read/Write, Type: enum

Select

select-none: Command, Arg list: (void).

select-atom: Command, Arg list: integer, integer.

select-residue: Command, Arg list: integer, integer.

un-select-atom: Command, Arg list: integer, integer.

un-select-residue: Command, Arg list: integer, integer.

Ask About Selection

selected-atom-count: Variable, Readonly, Type: integer.

selected-atom: Variable, Readonly, Type: vector of integer, integer.

selection-value: Variable, Readonly, Type: float.

Operate on Selection

delete-selected-atoms: Command, Arg list: (void).

reorder-selections: Variable, Read/Write, Type: boolean.

Named Selections

name-selection: Command, Arg list: string.

delete-named-selection: Command, Arg list: string.

named-selection-count: Variable, Readonly, Type: integer.

named-selection-name: Variable, Readonly, Type: vector of string.

named-selection-value: Variable, Readonly, Type: vector of float.

select-name: Command, Arg list: string.

Other

selection-color: Variable, Read/Write, Type: enum

File Operations

The following script commands deal with operations on the molecule files and the import/export files.

Molecule File

file-format: Variable, Read/Write, Type: string.

path: Variable, Read/Write, Type: string.

current-file-name: Variable, Readonly, Type: string.

open-file: Command, Arg list: string.

merge-file: Command, Arg list: string.

write-file: Command, Arg list: string.

delete-file: Command, Arg list: string.

Options

file-needs-saved: Variable, Read/Write, Type: boolean.

velocities-in-hin-file: Variable, Read/Write, Type: boolean.

view-in-hin-file: Variable, Read/Write, Type: boolean.

PDB File

connectivity-in-pdb-file: Variable, Read/Write, Type: boolean.

hydrogens-in-pdb-file: Variable, Read/Write, Type: boolean.

non-standard-pdb-names: Variable, Read/Write, Type: boolean.

Import/Export

import-dipole: Variable, Read/Write, Type: boolean.

import-ir: Variable, Read/Write, Type: boolean.

import-orbitals: Variable, Read/Write, Type: boolean.

import-property-file: Command, Arg list: string.

import-uv: Variable, Read/Write, Type: boolean.

export-dipole: Variable, Read/Write, Type: boolean.

export-ir: Variable, Read/Write, Type: boolean.

export-orbitals: Variable, Read/Write, Type: boolean.

export-property-file: Command, Arg list: string.

export-uv: Variable, Read/Write, Type: boolean.

Other

file-diff-message: Command, Arg list: string, string, string, string.

write-atom-map: Command, Arg list: string.

Scripts

The following script commands have to do with the process of scripting itself. They are used to open script files, control execution of script commands, or control the process of notification. Certain other scripts control the OMSG output of a query or manage the menu items inserted under the script menu. Finally, values of HSVs can be pushed and popped with a stack.

Script Files

read-script: Command, Arg list: string.

read-tcl-script: Command, Arg list: string.

compile-script-file: Command, Arg list: string, string.

read-binary-script: Command, Arg list: string.

Execution

query-value: Command, Arg list: .

execute-string: Command, Arg list: string.

pause-for: Command, Arg list: integer in range (0 .. 32767).

exit-script: Command, Arg list: (void).

Notifications

notify-on-update: Command, Arg list: string.

cancel-notify: Command, Arg list: string.

notify-with-text: Variable, Read/Write, Type: boolean.

variable-changed: Command, Arg list: string.

OMSGs

append-omsgs-to-file: Command, Arg list: string.

omsg-file: Variable, Read/Write, Type: string.

omsgs-not-to-file: Command, Arg list: (void).

omsgs-to-file: Command, Arg list: string.

query-response-has-tag: Variable, Read/Write, Type: boolean.

Menus

change-user-menuitem: Command, Arg list: integer, string, string.

script-menu-caption: Variable, Read/Write, Type: vector of string.

script-menu-checked: Variable, Read/Write, Type: vector of boolean.

script-menu-command: Variable, Read/Write, Type: vector of string.

script-menu-enabled: Variable, Read/Write, Type: vector of boolean.

script-menu-help-file: Variable, Read/Write, Type: vector of string.

script-menu-help-id: Variable, Read/Write, Type: vector of integer.

script-menu-in-use: Variable, Read/Write, Type: vector of boolean.

script-menu-message: Variable, Read/Write, Type: vector of string.

Stack Operation

pop-no-value: Command, Arg list: string.

pop-value: Command, Arg list: string.

push: Command, Arg list: string.

Other

execute-client: Command, Arg list: string.

execute-hyperchem-client: Command, Arg list: string.

message: Variable, Read/Write, Type: string.

one-line-arrays: Variable, Read/Write, Type: boolean.

Info

The following script commands are part of a capability for enquiring and obtaining information about a specific HSV.

info-access: Variable, Readonly, Type: string.

info-enum-id-of: Variable, Readonly, Type: string.

info-enum-list: Variable, Readonly, Type: string.

info-factory-setting: Variable, Readonly, Type: string.

info-id-of: Variable, Readonly, Type: integer.

info-type-of: Variable, Readonly, Type: string.

info-type-of-element: Variable, Readonly, Type: string.

info-variable-target: Variable, Read/Write, Type: string.

Errors

The following script commands deal with errors.

no-source-refs-in-errors: Command, Arg list: (void).

source-refs-in-errors: Command, Arg list: (void).

script-refs-in-errors: Variable, Read/Write, Type: boolean.

error: Variable, Read/Write, Type: string.

errors-are-not-omsgs: Command, Arg list: (void).

errors-are-omsgs: Command, Arg list: (void).

ignore-script-errors: Variable, Read/Write, Type: boolean.

hide-errors: Variable, Read/Write, Type: boolean.

Logging

The following script commands have to do with the process of creating a log file.

start-logging: Command, Arg list: string, boolean.

stop-logging: Command, Arg list: (void).

log-comment: Command. Arg list: string.

mechanics-print-level: Variable, Read/Write, Type: integer in range (0 .. 9).

quantum-print-level: Variable, Read/Write, Type: integer in range (0 .. 9).

Auxiliary

The following script commands defy simple classification.

Declarations

declare-integer: Command, Arg list: string.

declare-string: Command, Arg list: string.

Warnings

warning: Variable, Read/Write, Type: string.

warning-type: Variable, Read/Write, Type: enum(none, log, message).

hide-warnings: Variable, Read/Write, Type: boolean.

warnings-are-not-omsgs: Command, Arg list: void.

warnings-are-omsgs: Command, Arg list: void.

Screen Output

status-message: Variable, Read/Write, Type: string.

request: Command, Arg list: string.

Version

version: Variable, Readonly, Type: string.

serial-number: Variable, Readonly, Type: string.

Other

print-variable-list: Command, Arg list: string.

toggle: Command, Arg list: string.

Viewing

The following script commands are associated with the manipulations determining what one sees on the screen excluding the specific molecular rendering. Some of them are simply viewing transformations. Others move the molecules or show attributes of the molecules.

Alignment

align-molecule: Command, Arg list: list of enums

align-viewer: Command. Arg list: enum.

Redisplay

global-inhibit-redisplay: Variable, Readonly, Type: boolean.

inhibit-redisplay: Variable, Read/Write, Type: boolean.

Rotation

rotate-molecules: Command, Arg list: enum, float.

rotate-viewer: Command, Arg list: enum, float.

Translation

translate-selection: Command, Arg list: float, float, float.

translate-view: Command, Arg list: float, float, float.

translate-whole-molecules: Variable, Read/Write, Type: boolean.

translate-merged-systems: Variable, Read/Write, Type: boolean.

use-fast-translation: Variable, Read/Write, Type: boolean.

Window

window-height: Variable, Read/Write, Type: integer.

window-width: Variable, Read/Write, Type: integer.

Other

show-perspective: Variable, Read/Write, Type: boolean.

wall-eyed-stereo: Variable, Read/Write, Type: boolean.

zoom: Command, Arg list: float in range (0.01 .. 50).

show-axes: Variable, Read/Write, Type: boolean.

show-dipoles: Variable, Read/Write, Type: boolean.

Rendering

The following script commands affect the molecular rendering of the molecule in the workspace.

General Options

bond-spacing-display-ratio: Variable, R/W, Type: float in range (0 .. 1).

cpk-max-double-buffer-atoms: Variable, Read/Write, Type: integer.

dot-surface-angle: Variable, R/W, Type: float angle in range (-90 .. 90).

double-buffered-display: Variable, Read/Write, Type: boolean.

render-method: Variable, Read/Write, Type: enum.

Specific Rendering Options

balls-highlighted: Variable, R/W, Type: boolean

balls-radius-ratio: Variable, R/W, Type: float in range (0 .. 1).

balls-shaded: Variable, R/W, Type: boolean

cylinders-color-by-element: Variable, R/W, Type: boolean

cylinders-width-ratio: Variable, R/W, Type: float in range (0 .. 1).

spheres-highlighted: Variable, R/W, Type: boolean

spheres-shaded: Variable, R/W, Type: boolean

sticks-width: Variable, R/W, Type: integer in range (0 .. 25)

Show - Don't Show

show-hydrogen-bonds: Variable, Read/Write, Type: boolean.

show-hydrogens: Variable, Read/Write, Type: boolean.

show-isosurface: Command, Arg list: boolean.

show-multiple-bonds: Variable, Read/Write, Type: boolean.

show-periodic-box: Variable, Read/Write, Type: boolean.

show-ribbons: Variable, Read/Write, Type: boolean.

show-stereo: Variable, Read/Write, Type: boolean.

show-stereochem-wedges: Variable, Read/Write, Type: boolean.

show-vibrational-vectors: Variable, Read/Write, Type: boolean.

Coloring and Labeling

The following script commands affect the showing of labels for atoms or residues as well as the colors that appear on the screen for various objects.

Color

atom-color: Variable, Read/Write, Type: array of enum.

bond-color: Variable, Read/Write, Type: enum.

color-element: Command, Arg list: integer, enum.

color-selection: Command, Arg list: string.

negatives-color: Variable, Read/Write, Type: enum.

positives-color: Variable, Read/Write, Type: enum.

revert-element-colors: Command, Arg list: (void).

window-color: Variable, Read/Write, Type: enum.

Labels

atom-label-text: Variable, Readonly, Type: array of string.

atom-labels: Variable, Read/Write, Type: enum.

Images

The following script commands deal with getting bitmaps and metafiles (images) of molecules into the clipboard or into a file.

image-color: Variable, Read/Write, Type: boolean.

image-destination-clipboard: Variable, Read/Write, Type: boolean.

image-destination-file: Variable, Read/Write, Type: boolean.

image-file-bitmap: Variable, Read/Write, Type: boolean.

image-file-bitmapRGB: Variable, Read/Write, Type: boolean.

image-file-metafile: Variable, Read/Write, Type: boolean.

image-include-cursor: Variable, Read/Write, Type: boolean.

image-source-window: Variable, Read/Write, Type: enum.

Model Building

The following script commands deal with aspect of drawing and creating molecules with the model builder.

Options

allow-ions: Variable, Read/Write, Type: boolean.

explicit-hydrogens: Variable, Read/Write, Type: boolean.

default-element: Variable, Read/Write, Type: integer in range (0 .. 103).

Drawing

create-atom: Command, Arg list: integer in range (0 .. 103).

set-bond: Command, Arg list: integer, integer, integer, integer, enum.

delete-atom: Command, Arg list: integer, integer.

Constraints

constrain-geometry: Command, Arg list: string.

constrain-bond-length: Command, Arg list: float in range (0 .. 100).

constrain-bond-angle: Command, Arg list: float angle in range (-360 .. 360).

constrain-bond-torsion: Command, Arg list: angle in range (-360 ... 360)

unconstrain-bond-length: Command, Arg list: (void).

unconstrain-bond-angle: Command, Arg list: (void).

unconstrain-bond-torsion: Command, Arg list: (void).

Other

is-ring-atom: Variable, Readonly, Type: array of boolean.

neighbors: Variable, Readonly, Type: array of (unknown).

Stereochemistry

The following script commands deal with aspects of creating and showing specific stereochemistry.

builder-enforces-stereo: Variable, Read/Write, Type: boolean.

change-stereochem: Command, Arg list: integer, integer.

chirality: Variable, Read/Write, Type: array of string.

constrain-bond-down: Command, Arg list: integer, integer, integer, integer.

constrain-bond-up: Command, Arg list: integer, integer, integer, integer.

constrain-change-stereo: Command, Arg list: integer, integer.

constrain-fix-stereo: Command, Arg list: integer, integer.

cycle-atom-stereo: Command, Arg list: integer, integer.

cycle-bond-stereo: Command, Arg list: integer, integer, integer, integer.

remove-all-stereo-constraints: Command, Arg list: (void).

remove-stereo-constraint: Command, Arg list: integer, integer.

Atom Properties

The following script commands deal with changing or displaying properties of individual atoms perhaps associated with labels.

Labels

atom-charge: Variable, Read/Write, Type: array of float.

atom-mass: Variable, Read/Write, Type: array of float.

atom-name: Variable, Read/Write, Type: array of string.

atom-type: Variable, Read/Write, Type: array of string.

atomic-number: Variable, Read/Write, Type: array of integer.

atomic-symbol: Variable, Readonly, Type: array of string.

set-atom-charge: Command, Arg list: float in range (-100 .. 100).

set-atom-type: Command, Arg list: string.

Coordinates and Velocities

coordinates: Variable, Read/Write, Type: array of float, float, float.

set-velocity: Command, Arg list: enum, float, float, float.

velocities: Variable, Read/Write, Type: array of float, float, float

set-bond-angle: Command, Arg list: float angle in range (0 .. 180).

set-bond-length: Command, Arg list: float in range (0 .. 3200).

set-bond-torsion: Command, Arg list: float angle in range (-360 .. 360).

Other

coordination: Variable, Readonly, Type: array of integer.

Molecule Properties

The following script commands deal with some properties of molecules or the atoms in molecules.

Charge-Multiplicity

multiplicity: Variable, Read/Write, Type: integer in range (1 .. 6).

quantum-total-charge: Variable, R/W, Type: integer

Counts

atom-count: Variable, Readonly, Type: vector of integer.

molecule-count: Variable, Readonly, Type: integer.

Properties

dipole-moment: Variable, R/W , Type: float in range (-1e+10 .. 1e+10).

dipole-moment-components: Variable, R/W, Type: float, float, float.

heat-of-formation: Variable, R only , Type: float in range (-1e+10 .. 1e+10).

moments-of-inertia: Variable, Readonly, Type: float float float.

Back Ends

The following script commands deal with the operation of the computational back ends and communication with them.

Basic

backend-active: Variable, Read/Write, Type: boolean.

backend-communications: Variable, Read/Write, Type: enum.

Large Communication Structures

atom-info: Variable, Readonly, Type: (unknown).

mechanics-info: Variable, Readonly, Type: (unknown).

mechanics-data: Variable, Readonly, Type: (unknown).

Remote Back Ends

backend-host-name: Variable, Read/Write, Type: string.

backend-process-count: Variable, R/W, Type: integer in range (1 .. 32).

backend-user-id: Variable, Read/Write, Type: string.

backend-user-password: Variable, Read/Write. Type: string.

Molecular Mechanics Calculations

The following script commands deal with computations in molecular mechanics.

Method

molecular-mechanics-method: Variable, Read/Write, Type: enum.

is-extended-hydrogen: Variable, Readonly, Type: array of boolean.

keep-atom-charges: Variable, Read/Write, Type: boolean.

Energy Components

bend-energy: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

stretch-energy: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

torsion-energy: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

nonbond-energy: Variable, R only, Type: float in range (-1e+10 .. 1e+10).

estatic-energy: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

hbond-energy: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

Cutoffs

cutoff-type: Variable, Read/Write, Type: enum.

cutoff-inner-radius: Variable, Read/Write, Type: float in range (0 .. 1e+10).

cutoff-outer-radius: Variable, Read/Write, Type: float in range (0 .. 1e+10).

Scale Factors

mechanics-dielectric: Variable, R/W, Type: enum, enum, enum, enum.

mechanics-dielectric-scale-factor: Variable, R/W, Type: four floats.

mechanics-electrostatic-scale-factor: Variable, R/W, Type: four floats.

mechanics-mmp-electrostatics: Variable, Read/Write, Type: enum.

mechanics-van-der-waals-scale-factor: Variable, R/W, Type: four floats.

Parameters

parameter-set-changed: Variable, Read/Write, Type: boolean.

use-parameter-set: Command,Arg list: string.

Amino Acids and Nucleic Acids

The following script commands pertain to amino acid templates and the construction of peptides or to nucleic acid templates and the construction of DNA-like structures.

Amino Acids

add-amino-acid: Command, Arg list: string.

amino-alpha-helix: Command, Arg list: (void).

amino-beta-sheet: Command, Arg list: (void).

amino-isomer: Variable, Read/Write, Type: enum.

amino-omega: Variable, Read/Write, Type: float angle in range (-360 .. 360).

amino-phi: Variable, Read/Write, Type: float angle in range (-360 .. 360).

amino-psi: Variable, Read/Write, Type: float angle in range (-360 .. 360).

Nucleic Acids

add-nucleic-acid: Command, Arg list: string.

nucleic-a-form: Command, Arg list: (void).

nucleic-alpha: Variable, Read/Write, Type: float angle in range (-360 .. 360).

nucleic-b-form: Command, Arg list: (void).

nucleic-backwards: Variable, Read/Write, Type: boolean.

nucleic-beta: Variable, Read/Write, Type: float angle in range (-360 .. 360).

nucleic-chi: Variable, Read/Write, Type: float angle in range (-360 .. 360).

nucleic-delta: Variable, Read/Write, Type: float angle in range (-360 .. 360).

nucleic-double-strand: Variable, Read/Write, Type: boolean.

nucleic-epsilon: Variable, R/W, Type: float angle in range (-360 .. 360).

nucleic-gamma: Variable, R/W, Type: float angle in range (-360 .. 360).

nucleic-sugar-pucker: Variable, Read/Write, Type: enum.

nucleic-z-form: Command, Arg list: (void).

nucleic-zeta: Variable, Read/Write, Type: float angle in range (-360 .. 360).

General Residue

mutate-residue: Command, Arg list: string.

residue-charge: Variable, Readonly, Type: array of float.

residue-coordinates: Variable, Readonly, Type: array of float, float, float.

residue-count: Variable, Readonly, Type: vector of integer.

residue-label-text: Variable, Readonly, Type: array of string.

residue-labels: Variable, Read/Write, Type: enum.

residue-name: Variable, Readonly, Type: array of string.

Molecular Dynamics and Monte Carlo

The following script commands pertain to the two type of molecular dynamics (normal and Langevan) and the very closely related Mont eCarlo calculations.

Basic

do-molecular-dynamics: Command, Arg list: (void)

do-langevin-dynamics: Command, Arg list: (void)

do-monte-carlo: Command, Arg list: (void)

dynamics-restart: Variable, Read/Write, Type: boolean.

dynamics-average-period: Variable, R/W, Type: integer.

dynamics-collection-period: Variable, R/W, Type: integer.

screen-refresh-period: Variable, Read/Write, Type: integer.

Run Parameters

dynamics-bath-relaxation-time: Variable, R/W, Type: float.

dynamics-constant-temp: Variable, Read/Write. Type: boolean.

dynamics-cool-time: Variable, Read/Write, Type: float.

dynamics-heat-time: Variable, Read/Write, Type: float.

dynamics-final-temp: Variable, R/W, Type: float.

dynamics-friction-coefficient: Variable, R/W, Type: float

dynamics-run-time: Variable, Read/Write, Type: float.

dynamics-seed: Variable, Read/Write, Type: integer.

dynamics-simulation-temp: Variable, Read/Write, Type: float.

dynamics-starting-temp: Variable, Read/Write, Type: float.

dynamics-temp-step: Variable, Read/Write, Type: float.

dynamics-time-step: Variable, Read/Write, Type: float.

Averaging

append-dynamics-average: Command, Arg list: string.

append-dynamics-graph: Command, Arg list: string.

dynamics-info-elapsed-time: Variable, R only, Type: float.

dynamics-info-kinetic-energy: Variable, Readonly, Type: float.

dynamics-info-last-update: Variable, Readonly, Type: boolean.

dynamics-info-potential-energy: Variable, Readonly, Type: float.

dynamics-info-temperature: Variable, Readonly, Type: float.

dynamics-info-total-energy: Variable, Readonly, Type: float.

Playback

dynamics-playback: Variable, Read/Write, Type: enum.

dynamics-playback-end: Variable, Read/Write, Type: integer.

dynamics-playback-period: Variable, Read/Write, Type: integer.

dynamics-playback-start: Variable, Read/Write, Type: integer.

dynamics-snapshot-filename: Variable, Read/Write, Type: string.

dynamics-snapshot-period: Variable, Read/Write, Type: integer.

Monte Carlo Specific

monte-carlo-cool-steps: Variable, Read/Write, Type: float

monte-carlo-heat-steps: Variable, Read/Write, Type: float

monte-carlo-info-acceptance-ratio: Variable, Readonly, Type: float

monte-carlo-max-delta: Variable, R/W, Type: float

monte-carlo-run-steps: Variable, R/W, Type: float

Optimization

The following script commands deal with facets of geometry optimization.

Basic

do-optimization: Command, Arg list: (void).

optim-algorithm: Variable, Read/Write, Type: enum.

optim-converged: Variable, Readonly, Type: boolean.

optim-convergence: Variable, Read/Write, Type: float in range (0 .. 100).

optim-max-cycles: Variable, Read/Write, Type: integer in range (1 .. +Inf).

rms-gradient: Variable, Readonly, Type: float in range (-1e+10 .. 1e+10).

Restraints

restraint: Command, Arg list: string, float, float.

restraint-tether: Command, Arg list: complex.

use-no-restraints: Command, Arg list: (void).

use-restraint: Command, Arg list: string, boolean.

General Quantum Mechanics

The following script commands deal with facets of quantum mechanical computations that are independent of the quantum mechanical method, whether it is semi-empirical or ab initio.

Input Parameters

do-qm-calculation: Variable, Read/Write, Type: boolean.

uhf: Variable, Read/Write, Type: boolean.

accelerate-scf-convergence: Variable, Read/Write, Type: boolean.

alpha-orbital-occupancy: Variable, Read/Write, Type: vector of float.

beta-orbital-occupancy: Variable, Read/Write, Type: vector of float.

excited-state: Variable, Read/Write, Type: boolean.

max-iterations: Variable, Read/Write, Type: integer in range (1 .. 32767).

scf-convergence: Variable, Read/Write, Type: float in range (0 .. 100).

Output Results

orbital-count: Variable, Readonly, Type: integer.

alpha-scf-eigenvector: Variable, Read/Write, Type: vector of float-list.

beta-scf-eigenvector: Variable, Read/Write, Type: vector of float-list.

scf-orbital-energy: Variable, Read/Write, Type: vector of float.

orbital-results: Variable, Read/Write, Type: vector of float-list.

scf-electronic-energy: Variable, Readonly, Type: float.

Semi-empirical Calculations

The following script commands deal with the input parameters required to specifically perform semi-empirical calculations and with their results. Some of the input parameteters for semi-empirical methods are essentially part of the `*.abp` parameter files.

General

semi-empirical-method: Variable, Read/Write, Type: enum.

d-orbitals-on-second-row: Variable, Read/Write, Type: boolean.

scf-atom-energy: Variable, Readonly, Type: float.

scf-binding-energy: Variable, Readonly, Type: float.

scf-core-energy: Variable, Readonly, Type: float.

Huckel

huckel-constant: Variable, Read/Write, Type: float in range (0 .. 10).

huckel-scaling-factor: Variable, R/W, Type: float in range (0 .. 100000).

huckel-weighted: Variable, Read/Write, Type: boolean.

ZINDO

zindo-1-pi-pi: Variable, Read/Write, Type: float in range (0 .. 2).

zindo-1-sigma-sigma: Variable, Read/Write, Type: float in range (0 .. 2).

zindo-s-pi-pi: Variable, Read/Write, Type: float in range (0 .. 2).

zindo-s-sigma-sigma: Variable, Read/Write, Type: float in range (0 .. 2).

Ab Initio Calculations

The following script commands pertain to performing ab initio calculations - their inputs and their results.

Input Options

abinitio-calculate-gradient: Variable, Read/Write, Type: boolean.

abinitio-d-orbitals: Variable, Read/Write, Type: boolean.

abinitio-f-orbitals: Variable, Read/Write, Type: boolean.

abinitio-direct-scf: Variable, Read/Write, Type: boolean.

abinitio-mo-initial-guess: Variable, Read/Write, Type: enum.

abinitio-mp2-correlation-energy: Variable, Read/Write, Type: boolean.

abinitio-mp2-frozen-core: Variable, Read/Write, Type: boolean.

abinitio-scf-convergence: Variable, R/W, Type: float in range (0 .. 100).

abinitio-use-ghost-atoms: Variable, Read/Write, Type: boolean.

Basis Set

assign-basisset: Command, Arg list: string.

atom-basisset: Variable, Read/Write, Type: array of string.

atom-extra-basisset: Variable, Read/Write, Type: array of string, float.

basisset-count: Variable, Readonly, Type: integer.

2-electron Integrals

abinitio-buffer-size: Variable, Read/Write, Type: integer.

abinitio-cutoff: Variable, Read/Write, Type: float in range (0 .. 1e+10).

abinitio-integral-format: Variable, Read/Write, Type: enum.

abinitio-integral-path: Variable, Read/Write, Type: string.

Results

mp2-energy: Variable, Readonly, Type: float.

Configuration Interaction

The following script commands are relevant to post-SCF configuration interaction calculations.

ci-criterion: Variable, Read/Write, Type: enum.

ci-excitation-energy: Variable, Read/Write, Type: float in range (0 .. 10000).

ci-occupied-orbitals: Variable, R/W, Type: integer in range (0 .. 32767).

ci-unoccupied-orbitals: Variable, R/W, Type: integer in range (0 .. 32767).

configuration-interaction: Variable, Read/Write, Type: enum.

Infrared Spectra

The following script commands are relevant to the calculation and display of vibrational spectra.

Animations

animate-vibrations: Variable, Read/Write, Type: boolean.

ir-animate-amplitude: Variable, Read/Write, Type: float in range (0 .. 10).

ir-animate-cycles: Variable, Read/Write, Type: integer in range (0 .. +Inf).

ir-animate-steps: Variable, Read/Write, Type: integer in range (3 .. +Inf).

Spectra

ir-band-count: Variable, Read/Write, Type: integer.

vibrational-mode: Variable, Read/Write, Type: integer.

ir-frequency: Variable, Read/Write, Type: vector of float.

ir-intensity: Variable, Read/Write, Type: vector of float.

ir-intensity-components: Variable, R/W, Type: vector of float, float, float.

ir-normal-mode: Variable, Read/Write, Type: vector of float-list.

UV Spectra

The following script commands are relevant to the calculation of electronic spectra.

configuration: Variable, Read/Write, Type: integer.

uv-band-count: Variable, Read/Write, Type: integer.

uv-dipole-components: Variable, Read/Write, Type: vector of float-list.

uv-energy: Variable, Read/Write, Type: vector of float.

uv-oscillator-strength: Variable, Read/Write, Type: vector of float.

uv-spin: Variable, Read/Write, Type: vector of float.

uv-total-dipole: Variable, Read/Write. Type: vector of float.

uv-transition-dipole: Variable, R/W, Type: vector of float, float, float.

Plotting

the following script commands are relevant to the plotting of 2D and 3D contours and renderings of orbitals, electron density, spin density and electrostatic potentials.

General Options

graph-beta: Variable, Read/Write, Type: boolean.

graph-data-type: Variable, Read/Write, Type: enum.

graph-orbital-selection-type: Variable, Read/Write, Type: enum.

graph-orbital-offset: Variable, Read/Write, Type: integer in range (0 .. +Inf).

2D

do-qm-graph: Variable, Read/Write, Type: boolean.

graph-contour-increment: Variable, Read/Write, Type: float.

graph-contour-increment-other: Variable, Read/Write, Type: boolean.

graph-contour-levels: Variable, Read/Write, Type: integer.

graph-contour-start: Variable, Read/Write, Type: float.

graph-contour-start-other: Variable, Read/Write, Type: boolean.

3D

do-qm-isosurface: Variable, Read/Write, Type: boolean.

isosurface-grid-step-size: Variable, R/W, Type: float in range (0 .. 1e+10)

isosurface-hide-molecule: Variable, Read/Write, Type: boolean

isosurface-map-function: Variable, Read/Write, Type: boolean

isosurface-map-function-display-legend: Variable, R/W, Type: boolean

isosurface-map-function-range: Variable, Read/Write, Type: float, float

isosurface-mesh-quality: Variable, Read/Write, Type: enum

isosurface-render-method: Variable, Read/Write, Type: enum

isosurface-threshold: Variable, Read/Write, Type: float.

isosurface-transparency-level: Variable, R/W, Type: float in range (0 .. 1)

isosurface-x-min: Variable, Read/Write, Type: float.

isosurface-x-nodes: Variable, R/W, Type: integer in range (0 .. 32767).

isosurface-y-min: Variable, Read/Write, Type: float.

isosurface-y-nodes: Variable, R/W, Type: integer in range (0 .. 32767).

isosurface-z-min: Variable, Read/Write, Type: float.

isosurface-z-nodes: Variable, R/W, Type: integer in range (0 .. 32767).

Grid

graph-data-row: Variable, Readonly, Type: vector of float-list.

graph-horizontal-grid-size: Variable, R/W, Type: integer in (2 .. 8192).

graph-plane-offset: Variable, Read/Write, Type: float.

graph-vertical-grid-size: Variable, R/W, Type: integer in range (2 .. 8192).

grid-max-value: Variable, Readonly, Type: float.

grid-min-value: Variable, Readonly, Type: float.

isosurface-grid-step-size: Variable, R/W, Type: float in range (0 .. 1e+10).

Appendix B
Listing of Tcl Commands

The Tcl Commands

In this appendix we give a very brief alphabetic list of the most important Tcl commands and functions. They are listed with their arguments, with optional arguments in angular brackets. However, no detailed description of each command is given here and a reference book on Tcl is strongly suggested.

abs (x)

acos (x)

append varName value <value …>

asin (x)

atan(x)

atan2 (x,y)

break

catch command <varName>

cd <dirName>

ceil (x)

close fileId

concat <list list …>

continue

cos (x)

cosh (x)

double (i)

eof fileId

error message <info> <code>

eval arg <arg arg …>

exec <-keepnewline> <- -> <arg ..>

exit <code>

exp (x)

expr arg <arg arg …>

file atime name

file dirname name

file executable name

file exists name

file extension name

file isdirectory name

file isfile name

file lstat name arrayName

file mtime name

file option name <arg arg …>

file owned name

file readable name

file readlink name

file rootname name

file size name

file stat name arrayName

file tail name

file type name

file writable name

floor (x)

flust file Id

fmod (x,y)

for init test reinit body

foreach varName list body

format formatString <value value …>

gets file Id <varName>

glob <-nocomplain> <- -> pattern <pattern …>

global name1 <name2 …>

hypot (x,y)

if test1 body1 <elseif test2 body2 elseif …> <else bodyn>

incr varName <increment>

int (x)

join list <joinString>

lappend varName value <value …>

lindex list index

linsert list index value <value …>

list <value value…>

llength list

log (x)

log10 (x)

lrange list first last

lreplace list first last <value value …>

lsearch <-exact> <-glob> <-regexp> list pattern

lsort <-ascii> <-integer> <-real> <-command command>\ <-increasing> <-decreasing> list

open l command <access>

open name <access>

pid <fileId>

pow (x,y)

proc name argList body

puts <-nonewline> <fileId> string

pwd

read <-nonewline> fileId

read fileId numBytes

regexp <-indices> <-nocase> <- -> exp string <matchVar> \ <subVar subVar …>

regsub <-all> <-nocase> <- -> exp string subSpec varName

return <-code code> <-errorinfo info> <-errorcode code> <string>

return <options> <value>

round (x)

scan string format varName <varName varName …>

seek fileId offset <origin>

set varName <value>

sin (x)

sinh (x)

source fileName

split string <splitChars>

sqrt (x)

string compare string1 string2

string first string1 string2

string index string charIndex

string last string1 string2

string length string

string match pattern string

string range string first last

string tolower string

string toupper stsring

string trim string <chars>

string trimleft string <chars>

string trimright striing <chars>

switch <options> string pattern body <pattern body …>

switch <options> string (pattern body <pattern body …>)

tan (x)

tanh (x)

tell fileId

unset varName <varName varName …>

uplevel <level> arg <arg arg …>

upvar <level> otherVar1 my Var1 <otherVar2 myVar2 …>

while test body

Appendix C
Classification of HAPI Calls

This appendix lists each of the HAPI calls and gives details as to their use, their declaration and their parameters. The HAPI calls are classified according to their use. Visual Basic Arguments listed as N/A can be inferred from the corrsponding C/C++ call.

The API functions

Functions for Initialization and Termination

The functions in this group are responsible for initialization of the HAPI library, establishing a connection with HyperChem and termination of the connection.

hcInitAPI

The function performs initialization of the HAPI.

API header

```
BOOL _stdcall hcInitAPI()
```

FORTRAN interface

```
logical function hfInitAPI()
```

VISUAL BASIC declaration

```
Declare Function hbInitAPI Lib "hapi.dll" Alias "hcInitAPI" () As Long
```

Parameters

The function takes no parameters.

Return Value

The function returns TRUE if initialization was successful. The function returns FALSE if the user application is already connected with HyperChem.

Remarks

There is no need for explicitly calling the function if you use any of the recommended methods described in the section "How to use the CDK API" of Chapter 11. Both LoadHAPI and automatic DLL initialization code call the function. However the call may be required if you perform your own Run-Time Load for an unusual application.

hcConnect

The function establishes a link between the user application and HyperChem. It must be called before ANY other function is called except for auxiliary functions.

API header

```
BOOL _stdcall hcConnect(LPSTR lszCmd);
```

FORTRAN interface

```
logical function hfConnect(init_string)
character*(*) init_string
```

VISUAL BASIC declaration

```
Declare Function hbConnect Lib "hapi.dll" Alias "hcConnect" (ByVal command As
String) As Long
```

Parameters

C/C++: *LPSTR lszCmd - command line received from HyperChem or empty string (""). See Remarks.*

FORTRAN: *character*(*) init_string - command line received from HyperChem or empty string (Fortran string). See Remarks.*

VB: *command - VB variable of string type*

Return Value

The function returns TRUE(1) when it finds HyperChem and establishes a connection to it. When it fails, it returns FALSE(0).

Remarks

The command line string lszCmd is received from HyperChem when the application is called via the 'execute-hyperchem-client' script command. The string contains the information required to connect the user application with a proper instance of HyperChem. When HyperChem initiates the user application after issuing a script command:

```
execute-hyperchem-client userapp.exe
```

the user application receives a command line parameter. The parameter has the form:

```
-hinst:ChemServer-xxxx
```

For example:

```
-hinst:ChemServer-510e
```

The user application should read the command line and simply pass it as a parameter to hcConnect. If the parameter is not passed (lszCmd points to an empty string ""), the HAPI connects to the first HyperChem instance it finds. This may cause problems when there are more than one active HyperChem on the desktop.

If the flag errACTION_MESS_BOX is raised for error processing (see Auxiliary functions) the HAPI displays a message box if the function is not able to establish a connection with HyperChem. The action then depends on the user's choice (Abort/Retry/Ignore).

When the user ignores the message, hcConnect returns FALSE. See Error processing section for details.

See Also

hcDisconnect, hcSetErrorAction, hcLastError

hcDisconnect

The function hcDisconnect closes the connection with HyperChem opened by hcConnect.

API header

```
BOOL hcDisconnect(void);
```

FORTRAN interface

```
logical function hfDisconnect()
```

VISUAL BASIC declaration

```
Declare Function hbDisconnect Lib "hapi.dll" Alias "hcDisconnect" () As Long
```

Parameters

The function takes no parameters

Return Value

The function returns TRUE if the disconnecting was successful.

Remarks

The function is called automatically when errors occurs and the user chooses ABORT as an action. The application should always close the connection before it exits, otherwise the number of open but not used DDE channels would grow unnecessarily.

See Also

hcConnect

hcExit

This function causes an immediate exit from the application that calls it.

API header

```
void hcExit(void);
```

FORTRAN interface

```
subroutine hfExit()
```

VISUAL BASIC declaration

```
Declare Sub hbExit Lib "hapi.dll" Alias "hcExit" ()
```

Parameters

The function takes no parameters.

Remarks

The function should be treated as an emergency exit rather than a regular method of exiting program operation.

Functions for Text-based Communication

hcExecTxt

This function executes a HyperChem command in a text format. The function is also used for updating HSV values using a text format.

API header

```
BOOL hcExecTxt(LPSTR script_cmd);
```

FORTRAN interface

```
logical function hfExecTxt(script_cmd)
character*(*) script_cmd
```

VISUAL BASIC declaration

```
Declare Function hbExecTxt Lib "hapi.dll" Alias "hcExecTxt" (ByVal script_cmd
As String) As Long
```

Parameters

C/C++: *The parameter is a NULL terminated string containing the command to be executed.*

FORTRAN: *The parameter is Fortran string containing the command to be executed.*

VB: *The parameter is VB string containing the command to be executed*

Return Value

The function returns TRUE (1) upon successful completion. If there is an error, the user is notified by the appropriate message-box. If the error processing level is set to errACTION_NO (see SetErrorAction), the user may call hcLastError for info about the error.

Remarks

The update of an HSV variable is conceptually similar to the execution of a command. Hence, to set the value of an HSV variable the user's application might use the following syntax:

```
result=hcExecTxt("coordinates(1,2)=0.1,0.2,0.3")
```

for setting the coordinates of atom 1 in molecule 2 to (0.1,0.2,0.3).

Some commands may take time to complete. The regular time-out value for command completion is set at about 65 seconds. The user may extend the time-out by calling hcSetTimeouts.

See Also

hcQueryTxt, hcExecBin

hcQueryTxt

The function queries for a value of an HSV variable in text mode.

API header

```
LPSTR hcQueryTxt(LPSTR hsv);
```

FORTRAN interface

```
logical function hfQueryTxt(hsv, res)
character*(*) hsv,res
```

VISUAL BASIC declaration

```
Declare Function hbQueryTxt Lib "hapi.dll" Alias "hcQueryTxt" (ByVal command
As String) As String
```

Parameters

C/C++: *Null terminated string containing HSV to be queried. May contain indices in the case of vector and array variables.*

FORTRAN: *'hsv' is a Fortran string containing the HSV to be queried. 'res' is a Fortran string where the output is placed.*

VB: *VB String containing HSV name to be queried.*

Return Value

In the C, C++, or VB version , the function allocates the memory required to hold the answer and returns a pointer or a NULL pointer if the answer is not valid. However, the Fortran interface receives logical .true. or .false. indicating successful or unsuccessful completion of the function.

Remarks

When accessed from C/C++, the function returns a pointer to the newly allocated memory. Hence, the calling application is responsible for deallocating this particular memory block after it has been used, using the hcFree call.

The Fortran interface deallocates the memory block automatically. However, it is the calling application's responsibility to pass a reference to the Fortran string ('res') of sufficient length. If the answer is longer than provided string, the output is truncated.

When Querying for elements of vectors and arrays, the regular 'parenthesis' syntax is used. For example:

txt_xyz=hcQueryTxt("coordinates(1,2)");

for querying the coordinates of atom 1 in molecule 2.

See Also

hcQueryBin; see remarks for hcExecTxt for information about modifying HSV values.

Functions for Binary Communication

Binary Execute and Query

hcExecBin

This function executes a binary representation of a script command.

API header

```
BOOL hcExecBin(HSV cmd, LPV args, DWORD args_length);
```

FORTRAN interface

```
logical function hfExecBin(cmd,args,args_length)
integer cmd,args,args_length
dimension args(args_length)
```

VISUAL BASIC declaration

```
Declare Function hbExecBin Lib "hapi.dll" Alias "hcExecBin" (ByVal cmd As Long,
ByRef args As IntBuff, ByVal args_length As Long) As Long
```

Parameters

C/C++: *'cmd' specifies the HSV code. The calling application should include an 'hsv.h' file to have the codes available. 'args' is a pointer to memory block containing possible arguments. 'args_length' is the total length of valid data pointed to by 'args'.*

FORTRAN: *The calling application should include an 'hsv.fi' file to have the relevant HSV codes available. The difference in the Fortran interface is that 'args' is the reference to integer array. When transfering arguments of different types, the user has to use an equivalence instruction to pack the array with the required data type.*

VB: *The 'cmd' and 'args_length' parameters have the same meaning as for C and*

Fortran. The 'args' parameter is of 'IntBuff' type. This user defined Visual Basic type is defined in 'hsv.bas' module file.

Return Value

The function returns TRUE when successfully completed.

Remarks

The function may be used to update the values of an HSV variable, similar to the update via the use of hcExecTxt. Data may need to be packed. However, it is highly recommended that you use instead the functions from the Binary Get/Set group, which perform all required packing internally.

Some commands may take time to complete. The regular time-out value for command completion is set at about 65 seconds. The user may extend the time-out by calling hcSetTimeouts

See Also

hcQueryBin, hcGetxxx and hcSetxxx functions

hcQueryBin

The function queries for the binary value of an HSV of any type.

API header

```
void* hcQueryBin(int hsv,int indx1,int indx2,int* resp_length);
```

FORTRAN interface

```
logical function hfQueryBin(hsv,indx1,indx2,result,resp_length)
integer hsv,indx1,indx2,resp_length,result
dimension result(resp_length)
```

VISUAL BASIC declaration

```
Declare Function hbQueryBin Lib "hapi.dll" Alias "hfQueryBin" (ByVal var As
Long, ByVal indx1 As Long, ByVal indx2 As Long, ByRef result As IntBuff, ByRef
cbL As Long) As Long
```

Parameters

C/C++: *'hsv' - the binary code for HSV to be queried. 'indx1', 'indx2' the indexes for vector and array HSVs conforming the rules: indx1=0, indx2=0 - queries scalar HSV, entire vector or entire array; indx1=n,indx2=0 - queries n-th element of the vector HSV; indx1=m, indx2=n - get m,n-th element of the array HSV. 'resp_length' is the pointer to an integer variable where the length of the memory allocated by the function and returned as a pointer is stored.*

FORTRAN: *The arguments: 'hsv', 'indx1' and 'indx2' have identical meaning as in C/C++ case. The fourth argument, 'result' is a reference to the integer array that will receive the result of the query. The fifth argument, 'resp_length' should contain maximum number of bytes that 'result' array can receive. Upon successful completition 'resp_length' will hold the number of bytes received.*

VB: *The 'var', 'indx1' and 'indx2' have the same meaning as for C/C++ and Fortran. 'result' is a variable of the 'IntBuff' type defined in hsv.bas. See remarks to hcExecBin for explanations. The fifth argument, 'resp_length' should contain maximum number of bytes that 'result' array can receive.*

Return Value

In the C/C++ case the function returns the pointer to the memory allocated to hold the result of the query. A null value indicates failure during the query operation. In the Fortran and VB interface the function returns 1 if it was completed successfully and 0 otherwise.

Remarks

In the C/C++ case the function allocates a required memory block and returns the resultant pointer. It is the calling application's responsibility to free the memory when it is no longer needed, by calling hcFree.

When querying for vector and array variable with indexes (indx1 and/or indx2 set to zero) the result of the query contains all of the elements of respective variable. If it is an array, the atom indices change faster than the molecule indices.

If, in the Fortran case, the size of 'result' is smaller than the size of the query results, only the first 'resp_length' bytes are actually transferred to the 'result' array.

See Also

hcExecBin, hcQueryTxt, hcGetxxx, hcSetxxx

Functions for Binary 'Get'

This group of HAPI functions provide the easiest way for querying (Get) and updating (Set) for HSV variables. The transfer is performed in binary mode, hence, it is the fastest method for exchanging data. The functions provide all the necessary low-level operations on the blocks of memory transferred between a user application and HyperChem. This isolates the user application from the tedious task of preparing and interpreting the binary messages used by a lower-level communication between HyperChem and the user's application.

hcGetInt

The function gets the value of a scalar HSV of integer type and returns that value.

API header

```
int hcGetInt(int hsv);
```

FORTRAN interface

```
integer function hfGetInt(hsv)
integer hsv
```

VISUAL BASIC declaration

```
Declare Function hbGetInt Lib "hapi.dll" Alias "hcGetInt" (ByVal hsv As Long)
As Long
```

Parameters

C/C++:	*HSV code of the variable*
FORTRAN:	*HSV code of the variable*
VB:	*HSV code of the variable*

Return Value

The function returns the value of the HSV variable of integer type.

Remarks

See Also

hcSetInt, Appendix A for the types of HSV variables

hcGetReal

The function gets the value of a scalar HSV of floating point (double precision) type and returns this value.

API header

```
double hcGetReal(HSV var);
```

FORTRAN interface

```
double precision function hfGetReal (hsv)
integer hsv
```

VISUAL BASIC declaration

```
Declare Function hbGetReal Lib "hapi.dll" Alias "hcGetReal" (ByVal hsv As Long)
As Double
```

Parameters

C/C++:	*HSV code of the variable*
FORTRAN:	*HSV code of the variable*
VB:	*HSV code of the variable*

Return Value

The function returns the double precision value of HSV variable.

Remarks

See Also

hcSetReal

hcGetIntVec

The function gets the contents of the HSV variable of integer vector type .

API header

```
int hcGetIntVec(HSV var, int* buff, int max_length);
```

FORTRAN interface

```
integer function hfGetIntVec (hsv, result, res_length)
integer hsv, result, res_length
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbGetIntVec Lib "hapi.dll" Alias "hcGetIntVec" (ByVal hsv As
Long, ByRef buff As IntBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the vector is copied, 'max_length' - the size of 'buff'*

FORTRAN: 'hsv' - the HSV code for the vector variable, 'result' - one dimensional integer array to receive the vector, 'res_length' - the size of 'result' array.

VB: *The 'hsv' and 'max_length' have the same meaning as in the C/C++ case. 'buff" is of IntBuff type defined in 'hsv.bas' module definition file.*

Return Value

The function returns the number of integer words transfered to the buffer, if the operation was completed successfuly, or 0 if it failed.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the amount of integer words in the HSV is larger than the

buffer, only the first 'max_length' words are copied into buffer. the same applies to the Fortran ‘result’ and ‘res_length’ variables.

See Also

hcSetIntVec, hcGetIntVecElm, hcGetIntArray, hcSetIntArray

hcGetIntArr

The function gets the contents of the HSV variable of the integer array type.

API header

```
int hcGetIntArr(HSV var, int* buff, int max_length);
```

FORTRAN interface

```
integer function hfGetIntArr (hsv, result, res_length)
integer hsv, result, res_length
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbGetIntArr Lib "hapi.dll" Alias "hcGetIntArr" (ByVal var As
Long, ByRef buff As IntBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the array is copied, 'max_length' - the size of 'buff'*

FORTRAN: ‘hsv’ - the HSV code for the array variable, ‘result’ - one dimensional integer array to receive the HSV array, ‘res_length’ - the size of the ‘result’ array.

VB: *N/A*

Return Value

The function returns the number of integer words transferred to the buffer, if the operation was completed successfuly, or 0 if it failed.

Remarks

The function does not allocate memory for 'buff' - it assumes that the

appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the amount of integer words in the HSV is larger than the buffer, only first 'max_length' words are copied into buffer. The same applies to the Fortran 'result' and 'res_length' variables.

See Also

hcGetIntArrElm, hcSetIntArr, hcSetIntArrElm

hcGetIntArrElm

The function gets the contents of the HSV variable of integer array type.

API header

```
int hcGetIntArrElm(HSV var, int atom_index, int molecule_index);
```

FORTRAN interface

```
integer function hfGetIntArrElm( hsv, atom_index, mol_index)
integer hsv, atom_index, mol_index
```

VISUAL BASIC declaration

```
Declare Function hbGetIntArrElm Lib "hapi.dll" Alias "hcGetIntArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array.*

FORTRAN: 'hsv' - the HSV code for the array variable, *'atom_index' - index for the row of array, 'mol_index' - index for the column of the array.*

VB: *N/A*

Return Value

The function returns the integer value of the element of the array HSV variable of type integer.

Remarks

HSV arrays are always the arrays with atom-in-molecule and molecule index. Both indices run from 1 to number of respective elements.

See Also

hcSetIntArrElm, hcGetInt, hcGetIntArr, hcSetInt, hcSetIntArr

hcGetRealVec

The function gets the contents of the HSV variable of the real (double precision) vector type.

API header

```
int hcGetRealVec(HSV var, double* buff,int max_length);
```

FORTRAN interface

```
integer function hfGetRealVec (hsv, result, res_length)
integer hsv, res_length
double precision result
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbGetRealVec Lib "hapi.dll" Alias "hcGetRealVec" (ByVal var
As Long, ByRef buff As DblBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the vector is copied, 'max_length' - the size of 'buff'*

FORTRAN: ‘hsv’ - the HSV code for the vector variable, ‘result’ - one dimensional double precision array to receive the HSV vector, ‘res_length’ - the size of ‘result’ array.

VB: *N/A*

Return Value

The function returns the number of double precision words transfered to the buffer, if the operation was completed successfully, or 0 if it failed.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the amount of double precision words in the HSV is larger than the buffer, only first 'max_length' words are copied into buffer. the same applies to Fortran 'resul' and 'res_length' variables.

See Also

hcGetRealVecElm, hcSetRealVec, hcGetReal, hcSetReal, hcGetRealArrXYZ, hcSetRealArrXYZ

hcGetRealArr

The function gets the contents of the HSV variable of the real (double precision) array type.

API header

```
int hcGetRealArr(HSV var, double* buff,int max_length);
```

FORTRAN interface

```
integer function hfGetRealVec (hsv, result, res_length)
integer hsv, res_length
double precision result
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbGetRealArr Lib "hapi.dll" Alias "hcGetRealArr" (ByVal var
As Long, ByRef buff As DblBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the array is copied, 'max_length' - the size of 'buff'*

FORTRAN: 'hsv' - the HSV code for the vector variable, 'result' - one dimensional double precision array to receive the HSV vector, 'res_length' - the size of 'result' array.

VB: N/A

Return Value

The function returns the number of double precision words transfered to the buffer, if the operation was completed successfuly, or 0 if it failed.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the amount of double precision words in the HSV is larger than the buffer, only first 'max_length' words are copied into buffer. The same applies to Fortran 'result' and 'res_length' variables.

See Also

hcGetRealArrElm, hcSetRealArr, hcSetRealArrElm, hcSetReal, hcGetReal

hcGetIntVecElm

The function gets the integer value of an element of the vector-type HSV variable.

API header

```
int hcGetIntVecElm(HSV var,int index);
```

FORTRAN interface

```
integer function hfGetIntVecElm (hsv, index)
integer hsv, index
```

VISUAL BASIC declaration

```
Declare Function hbGetIntVecElm Lib "hapi.dll" Alias "hcGetIntVecElm" (ByVal
var As Long, ByVal index As Long) As Long
```

Parameters

C/C++:	*'var' - the HSV code, 'index' - points out the element to be retrieved.*
FORTRAN:	‘hsv’ - the HSV code for the vector variable, ‘index’ - points out the element to be retrieved.
VB:	*N/A*

Return Value

The function returns the integer value of the element of an HSV variable of integer type.

Remarks

The first element of the vector is referred to as an element with index equal to one.

See Also

hcGetInt, hcSetInt, hcSetIntVecElm, hcGetIntVecElm

hcGetRealVecElm

The function gets the value of a double-precision element of a vector HSV variable..

API header

```
double hcGetRealVecElm(HSV var,int index);
```

FORTRAN interface

```
double precision function hfgetRealVecElm (hsv, index)
integer hsv, index
```

VISUAL BASIC declaration

```
Declare Function hbGetRealVecElm Lib "hapi.dll" Alias "hcGetRealVecElm" (ByVal
var As Long, ByVal index As Long) As Double
```

Parameters

C/C++:	*'var' - the HSV code, 'index' - points out the element to be retrieved.*
FORTRAN:	'hsv' - the HSV code for the vector variable, 'index' - points out the element.
VB:	*N/A*

Return Value

The function returns the double precision value of the element of the HSV vector of real type.

Remarks

The first element of the vector is refered to as an element with index equal to one.

See Also

hcSetRealVecElm, hcGetReal, hcSetReal, hcGetRealVec, hcSetRealVec

hcGetRealArrElm

The function gets an element of an array HSV of real (double precision type).

API header

```
double  hcGetRealArrElm(HSV var,int atom_index,int molecule_index);
```

FORTRAN interface

```
integer function hfgetRealArrElm( hsv, atom_index, molecule_index)
integer hsv, atom_index, molecule_index
```

VISUAL BASIC declaration

```
Declare Function hbGetRealArrElm Lib "hapi.dll" Alias "hcGetRealArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long) As Double
```

Parameters

C/C++:	*'var' - the HSV code, 'atom_index' - index for the row of array,*

'molecule_index' - index for the column of the array

FORTRAN: 'hsv' - the HSV code for the array variable, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array.

VB: *N/A*

Return Value

The function returns the double precision value of the element of the array HSV variable of real type.

Remarks

HSV arrays are always arrays with atom-in-molecule and molecule index. Both indices run from 1 to number of respective elements.

See Also

hcSetRealArrElm, hcGetReal, hcGetRealArr, hcSetReal, hcSetRealArr

hcGetRealArrXYZ

The function gets the three reals (double precision) that form an element of the array HSV variable. It's designed almost exclusively for getting Cartesian coordinates of an atom.

API header

```
int hcGetRealArrXYZ(HSV var,int atom_index,int molecule_index, double* x,
double* y, double* z);
```

FORTRAN interface

```
logical function hfGetrealArrXYZ (hsv, atom_index, molecule_index, x, y, z)
integer hsv, atom_index, molecule_index
double precision x, y, z
```

VISUAL BASIC declaration

```
Declare Function hbGetRealArrXYZ Lib "hapi.dll" Alias "hcGetRealArrXYZ" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByRef x
As Double, ByRef y As Double, ByRef z As Double) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'x' , 'y' and 'z' pointers to double precision words where the results will be placed.*

FORTRAN: 'hsv' - the HSV code, 'atom_index' - index for the row of the array, 'molecule_index' - index for the column of the array, 'x', 'y', and 'z' - double precision variables that receive the results.

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

The coordinates of atoms are represented in HyperChem as an array indexed bythe atom-in-molecule number and the molecule number. However the element of the array is not a number but a triple of numbers representing three cartesian components of the position of the atom in space. Both atom and molecule indices run from 1 tothe respective number of elements.

See Also

hcSetRealArrXYZ, hcGetRealArr, hcSetRealArr

hcGetRealVecXYZ

The function gets the three components of the elements of a vector HSV variable.

API header

```
int hcGetRealVecXYZ(HSV var,int atom_index,double* x, double* y, double* z);
```

FORTRAN interface

```
logical function hfGetRealVex\cXYZ (hsv, index, x, y, z)
integer hsv, index
double precision x, y, z
```

VISUAL BASIC declaration

```
Declare Function hbGetRealVecXYZ Lib "hapi.dll" Alias "hcGetRealVecXYZ" (ByVal
var As Long, ByVal index As Long, ByRef x As Double, ByRef y As Double, ByRef
z As Double) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'index' - index for the element of the vector, 'x' , 'y' and 'z' pointers to double precision words where the results will be placed.*

FORTRAN: 'hsv' - the HSV code, 'index' - index of the vector element, 'x', 'y', and 'z' - double precision values that receive the results.

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

Some properties in HyperChem are represented as a vector of triples usually representing three cartesian components of the property. The index runs from 1 to the respective number of elements.

See Also

hcSetRealVecXYZ, hcGetReal, hcSetReal, hcGetRealVec, hcSetRealVec

hcGetStr

This function retrives the contents of the HSV variable of string type.

API header

```
int hcGetStr(HSV var,char* buff,int max_length);
```

FORTRAN interface

```
integer function hfGetStr( hsv, result, res_length)
cahracter*(*) result
```

VISUAL BASIC declaration

```
Declare Function hbGetStr Lib "hapi.dll" Alias "hcGetStr" (ByVal var As Long,
ByVal buff As String, ByVal max_length) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the string is copied, 'max_length' - the size of 'buff'*

FORTRAN: 'hsv' - the HSV code, 'result' - character array where the string is copied, 'res_length' - the length of the 'result' string.

VB: *N/A*

Return Value

The function returns the number of charactesr copied into the buffer if the operation was sucessful or zero otherwise.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the number of characters in the HSV is larger than the buffer, only first 'max_length' bytes are copied into buffer.

See Also

hcSetStr, hcSetBlock, hcGetBlock

hcGetStrVecElm

The function retrives the contents of the element of the vector HSV variable of string type.

API header

```
int hcGetStrVecElm(HSV var, int index, char* buff, int max_length);
```

FORTRAN interface

```
integer function hfGetStrVecElm( hsv, index, result, res_length)
integer hsv, index, res_length
```

```
character*(*) result
```

VISUAL BASIC declaration

```
Declare Function hbGetStrVecElm Lib "hapi.dll" Alias "hcGetStrVecElm" (ByVal
var As Long, ByVal index As Long, ByVal buff As String, ByVal max_length As
Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'index' - points out the element to be retrieved, 'buff' pointer to the buffer where the string is copied, 'max_length' - the size of 'buff'*

FORTRAN: ‘hsv’ - the HSV code, ‘index’ - pointsout the element to be retreived, ‘result’ - character array where the string is copied, ‘res_length’ - the length of’result’ array.

VB: *N/A*

Return Value

The function returns the number of characters copied into the buffer if the operation was sucessful or zero otherwise.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the number of characters in the HSV is larger than the buffer, only first 'max_length' bytes are copied into buffer.

The first element of the vector has index equal to one.

See Also

hcSetStrVecElm, hcGetStr, hcSetStr

hcGetStrArrElm

The function retrives the contents of the element of the array HSV variable of string type.

API header

```
int hcGetStrArrElm(HSV var, int atom_index, int molecule_index, char* buff,
int max_length);
```

FORTRAN interface

```
integer function hfGetStrArrElm(hsv,atom_index,mol_index,result,res_length)
integer hsv, atom_index, mol_index, res_length
character*(*) result
```

VISUAL BASIC declaration

```
Declare Function hbGetStrArrElm Lib "hapi.dll" Alias "hcGetStrArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByVal buff
As String, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'buff' pointer to the buffer where the string is copied, 'max_length' - the size of 'buff'*

FORTRAN: ‘hsv’ - the HSV code, ‘atom_index’ - index for the row of array, ‘mol_index’ - index for the column of array, ‘result’ - character array where the string is copied, ‘res_length’ - the length of’result’ array.

VB: *N/A*

Return Value

The function returns the number of characters copied into buffer if the operation was sucessful or zero otherwise.

Remarks

The function does not allocate memory for 'buff' - it assumes that the appropriate memory block pointed out by 'buff' was allocated and is of 'max_length' size. If the number of characters in the HSV is larger than the buffer, only first 'max_length' bytes are copied into buffer.

Both indices start with one.

See Also

hcSetStrArrElm, hcSetStr, hcGetStr

hcGetBlock

The function retrives the contents of the whole HSV variable irrespetiv of its type.

API header

```
int hcGetBlock(HSV var, char* buff, int max_length);
```

FORTRAN interface

```
integer function hfgetBlock( hsv, result, res_length)
integer*1 result
```

VISUAL BASIC declaration

```
Declare Function hbGetBlock Lib "hapi.dll" Alias "hcGetBlock" (ByVal var As
Long, ByVal buff As String, ByVal max_length) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the data is copied, 'max_length' - the size of 'buff'*

FORTRAN: 'hsv' - the HSV code, 'result' - the integer*1 (byte) array where the data is copied, 'res_length' - the size of the result.

VB: *N/A*

Return Value

The function returns the number of bytes copied into the buffer.

Remarks

Some HSV variables, particularly some vectors and arrays, have the type of element which is not just an integer, real or string. The hcGetBlock was provided to get access to that kind of variable. However, the user's application is responsible for the interpretation of the data in the block and the proper "sorting-out" of individual elements of the block.

See Also

hcSetBlock, hcQueryBin

Functions for Binary ‘Set’

hcSetInt

The function updates the scalar HSV of integer type.

API header

```
int hcSetInt(int var,int value);
```

FORTRAN interface

```
logical function hfSetInt( hsv, value)
integer hsv, value
```

VISUAL BASIC declaration

```
Declare Function hbSetInt Lib "hapi.dll" Alias "hcSetInt" (ByVal hsv As Long,
ByVal value As Long) As Long
```

Parameters

C/C++: *‘var’ - HSV code of the variable to be modified, ‘value’ - new value for the variable.*

FORTRAN: *‘hsv’ - HSV code of the variable to be modified, ‘value’ - new value for the variable.*

VB: *‘var’ - HSV code of the variable to be modified, ‘value’ - new value for the variable.*

Return Value

The function returns 1 if the update was successful or 0 if it failed.

Remarks

See Also

hcGetInt, Appendix A for the types of HSV variables

hcSetReal

The function updates the scalar HSV of floating point (double precision) type.

API header

```
int hcSetReal(int var,double value);
```

FORTRAN interface

```
logical function hfSetReal( hsv, value)
integer hsv
double precision value
```

VISUAL BASIC declaration

```
Declare Function hbSetReal Lib "hapi.dll" Alias "hcSetReal" (ByVal hsv As Long,
ByVal value As Double) As Long
```

Parameters

C/C++: *'var' - HSV code of the variable to be modified, 'value' - new value for the variable.*

FORTRAN: *'hsv' - HSV code of the variable to be modified, 'value' - new value for the variable.*

VB: *'var' - HSV code of the variable to be modified, 'value' - new value for the variable.*

Return Value

The function returns 1 if the update was successful or 0 if it failed.

Remarks

See Also

hcGetReal

hcSetIntVec

The function updates the vector HSV variable of the integer element type.

API header

```
int hcSetIntVec(HSV var, int* buff, int length);
```

FORTRAN interface

```
logical function hfSetIntVec( hsv, result, res_length)
integer hsv, result, res_length
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbSetIntVec Lib "hapi.dll" Alias "hcSetIntVec" (ByVal hsv As
Long, ByRef buff As IntBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer with the new contens for the vector, 'length' - the number of elements in the 'buff' buffer.*

FORTRAN: 'hsv' - the HSV code for the vector variable, 'result' - one dimensional integer array containing data for the vector, 'res_length' - the size of 'result' array.

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

The user's application is responsible for the appropriate number of elements in the buffer. If the number was incorrect, HyperChem would signal an error.

See Also

hcGetIntVec, hcSetIntVecElm, hcSetIntArray, hcGetIntArray, hcSetIntVecElm

hcSetIntArr

The function updates the array HSV variable of the integer element type.

API header

```
int hcSetIntArr(int var, int* buff, int length);
```

FORTRAN interface

```
logical function hfSetIntArr( hsv, result, res_length)
integer hsv, result, res_length
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbSetIntArr Lib "hapi.dll" Alias "hcSetIntArr" (ByVal var As
Long, ByRef buff As IntBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer with the new contens for the array, 'length' - the number of elements in the 'buff' buffer.*

FORTRAN: 'hsv' - the HSV code for the array variable, 'result' - one dimensional integer array containing data for the array, 'res_length' - the size of 'result' array.

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

The user's application is responsible for the appropriate number of elements in the buffer. If the number was incorrect, HyperChem would signal an error.

See Also

hcSetIntArrElm, hcGetIntArr, hcGetIntArrElm

hcSetIntArrElm

The function updates the element of the array HSV variable of the integer array type.

API header

```
int hcSetIntArrElm(HSV var, int atom_index, int molecule_index, int value);
```

FORTRAN interface

```
logical function hfSetIntArrElm( hsv, atom_index, mol_index, value)
integer hsv, atom_index, mol_index, value
```

VISUAL BASIC declaration

```
Declare Function hbSetIntArrElm Lib "hapi.dll" Alias "hcSetIntArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByVal
value As Double) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'value' - the new value for the variable.*

FORTRAN: *'hsv' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'value' - the new integer value for the variable.*

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

HSV arrays are always the arrays with atom-in-molecule and molecule index. Both indices run from 1 to number of respective elements.

See Also

hcGetIntArrElm, hcSetInt, hcSetIntArr, hcGetInt, hcGetIntArr

hcSetRealVec

The function updates the contents of the HSV variable of the real (double precision) type.

API header

```
int hcSetRealVec(int var, double* buff,int length);
```

FORTRAN interface

```
logical function hfSetRealVec( hsv, result, res_length)
integer hsv, res_length
double precision result
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbSetRealVec Lib "hapi.dll" Alias "hcSetRealVec" (ByVal var
As Long, ByRef buff As DblBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer with the new contens for the vector, 'length' - the number of elements (double precission words) in the 'buff' buffer.*

FORTRAN: 'hsv' - the HSV code for the vector variable, 'result' - one dimensional double precision array containing the data for the vector, 'res_length' - the size of the 'result' array.

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

The user's application is responsible for the appropriate number of elements in the buffer. If the number was incorrect, HyperChem would signal an error.

See Also

hcSetRealVecElm, hcGetRealVec, hcSetReal, hcGetReal, hcSetRealArrXYZ, hcGetRealArrXYZ

hcSetRealArr

The function updates the HSV variable of the real (double precision) type.

API header

```
hcSetRealArr(int var, double* buff,int length);
```

FORTRAN interface

```
logical function hfSetRealArr( hsv, result, res_length)
integer hsv, res_length
double precision result
dimension result(res_length)
```

VISUAL BASIC declaration

```
Declare Function hbSetRealArr Lib "hapi.dll" Alias "hcSetRealArr" (ByVal var
As Long, ByRef buff As DblBuff, ByVal max_length As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer with the new contens for the array, 'length' - the number of elements (double precission words) in the 'buff' buffer.*

FORTRAN: ‘hsv’ - the HSV code for the array variable, ‘result’ - one dimensional double precision array containing the data for the array, ‘res_length’ - the size of the ‘result’ array.

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

The user's application is responsible forthe appropriate number of elements in the buffer. If the number was incorrect, HyperChem would signal an error.

See Also

hcSetRealArrElm, hcGetRealArr, hcGetRealArrElm, hcGetReal, hcSetReal

hcSetIntVecElm

The function updates the element of the vector HSV variable of integer type.

API header

```
int hcSetIntVecElm(int var,int index,int value);
```

FORTRAN interface

```
logical function hfSetIntVecElm( hsv, index, value)
integer hsv, index, value
```

VISUAL BASIC declaration

```
Declare Function hbSetIntVecElm Lib "hapi.dll" Alias "hcSetIntVecElm" (ByVal
var As Long, ByVal index As Long, ByVal value As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'index' - points out the element to be updated, 'value' - new value for the element.*

FORTRAN: *'hsv' - the HSV code fo rthe vector variable, 'index' - points out the element to be updated, 'value' - new value for the element.*

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

The first element of the vector has an index equal to one.

See Also

hcSetInt, hcGetInt, hcGetIntVecElm, hcSetIntVecElm

hcSetRealVecElm

The function updates the element of the vector HSV variable of double-precission type.

API header

```
int hcSetRealVecElm(int var,int index,double value);
```

FORTRAN interface

```
logical function hfSetRealVecElm( hsv, index, value)
integer hsv, index
double precision value
```

VISUAL BASIC declaration

```
Declare Function hbSetRealVecElm Lib "hapi.dll" Alias "hcSetRealVecElm" (ByVal
var As Long, ByVal index As Long, ByVal value As Long) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'index' - points out the element to be updated, 'value' - new value for the element.*

FORTRAN: *'hsv' - the HSV code for the vector variable, 'index' - points out the element to be updated, 'value' - new double precision value for the element.*

VB: *N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was

not updated.

Remarks

The first element of the vector has index equal to one.

See Also

hcGetRealVecElm, hcSetReal, hcGetReal, hcSetRealVec, hcGetRealVec

hcSetRealArrElm

The function updates the element of the array HSV variable of real (double precission type).

API header

```
int hcSetRealArrElm(int var,int atom_index,int molecule_index,double value);
```

FORTRAN interface

```
logical function hfSetRealVecElm( hsv, atom_index, mol_index, value)
integer hsv, atom_index, mol_index
double precision value
```

VISUAL BASIC declaration

```
Declare Function hbSetRealArrElm Lib "hapi.dll" Alias "hcSetRealArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByVal
value As Double) As Double
```

Parameters

C/C++:	*'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, value - the new value for the variable*
FORTRAN:	*'hsv' - the HSV code, 'atom_index' - index for the row of array, 'mol_index' - index for the column of the array, value - the new value for the variable*
VB:	*N/A*

Return Value

The function returns 1 if the variable was updated successfully, or 0 if it was not updated.

Remarks

HSV arrays are always arrays with an 'atom-in-molecule' and a'molecule' index. Both indices run from 1 to the number of respective elements.

See Also

hcGetRealArrElm, hcSetReal, hcSetRealArr, hcGetReal, hcGetRealArr

hcSetRealArrXYZ

The function updates an element of array HSV variable that has three real numbers as the element type. It's designed almost exclusively for updating the cartesian coordinates for an atom

API header

```
int hcSetRealArrXYZ)(int var,int atom_index,int molecule_index,double x,
double y, double z);
```

FORTRAN interface

```
logical function hfSetRealArrXYX( hsv, atom_index, mol_index, x, y, z)
integer hsv, atom_index, mol_index
double precision x, y, z
```

VISUAL BASIC declaration

```
Declare Function hbSetRealArrXYZ Lib "hapi.dll" Alias "hcSetRealArrXYZ" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByVal x
As Double, ByVal y As Double, ByVal z As Double) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'x' , 'y' and 'z' are double precision new values for the element.*

FORTRAN: *'hsv' - the HSV code, 'atom_index' - index for the row of array, 'mol_index' -*

index for the column of the array, 'x' , 'y' and 'z' are double precision new values for the element.

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

The coordinates of atoms are represented in HyperChem as an array indexed by the atom-in-molecule number and the molecule number. However the element of the array is not a number but a triple of numbers representing three Cartesian components of the position of the atom in space. Both atom and molecule indices run from 1 to the respective number of elements.

See Also

hcGetRealArrXYZ, hcSetRealArr, hcGetRealArr

hcSetRealVecXYZ

The function updates an element of vector HSV variable that has three real numbers as the element type.

API header

```
int hcSetRealVecXYZ(int var,int index,double x, double y, double z);
```

FORTRAN interface

```
logical function hfSetRealVecXYX( hsv, index, x, y, z)
integer hsv, index
double precision x, y, z
```

VISUAL BASIC declaration

```
Declare Function hbSetRealVecXYZ Lib "hapi.dll" Alias "hcSetRealVecXYZ" (ByVal
var As Long, ByVal index As Long, ByVal x As Double, ByVal y As Double, ByVal
z As Double) As Long
```

Parameters

C/C++:	*'var' - the HSV code, 'index' - index for the element of the vector, 'x' , 'y' and 'z' double precision new values for the element.*
FORTRAN:	*'hsv' - the HSV code, 'index' - index for the element of the vector, 'x' , 'y' and 'z' double precision new values for the element.*
VB:	*N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

Some properties in HyperChem are represented as the vector of triples, usually representing three Cartesian components of the property. The index runs from 1 to the respective number of elements.

See Also

hcGetRealVecXYZ, hcSetReal, hcGetReal, hcSetRealVec, hcGetRealVec

hcSetStr

The function updates the content of the HSV variable of string type.

API header

```
int hcSetStr(int var,char* string);
```

FORTRAN interface

```
logical function hfSetStr( hsv, buff)
integer hsv
character*(*) buff
```

VISUAL BASIC declaration

```
Declare Function hbSetStr Lib "hapi.dll" Alias "hcSetStr" (ByVal var As Long,
ByVal buff As String) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'string' pointer to the NULL-terminated string containing a new string.*

FORTRAN: *'hsv' - the HSV code, 'buff' - Fortran character array containing the new value of the string.*

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

See Also

hcGetStr, hcGetBlock, hcSetBlock

hcSetStrVecElm

The function updates the content of the element of the vector HSV variable of string type.

API header

```
int hcSetStrVecElm(int var, int index, char* string);
```

FORTRAN interface

```
logical function hfSetStrVecElm( hsv, index, buff)
integer hsv, index
character*(*) buff
```

VISUAL BASIC declaration

```
Declare Function hbSetStrVecElm Lib "hapi.dll" Alias "hcSetStrVecElm" (ByVal
var As Long, ByVal index As Long, ByVal buff As String, ByVal max_length As
Long, ByVal buff As String) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'index' - points out the element of the variable to be*

updated, 'string' pointer to the NULL-terminated string containing a new string.

FORTRAN: *'hsv' - the HSV code, 'index' - points out the element of the variable to be updated, 'buff' - character array containing the Fortran string.*

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

The first element of the vector has index equal to one.

See Also

hcGetStrVecElm, hcSetStr, hcGetStr

hcSetStrArrElm

The function updates the element of the array HSV variable of string type.

API header

```
int hcSetStrArrElm(int var,int atom_index, int molecule_index, char* string);
```

FORTRAN interface

```
logical function hfSetStrArrElm( hsv, atom_index, mol_index, buff)
integer hsv, atom_index, mol_index
character*(*) buff
```

VISUAL BASIC declaration

```
Declare Function hbSetStrArrElm Lib "hapi.dll" Alias "hcSetStrArrElm" (ByVal
var As Long, ByVal atom_index As Long, ByVal molecule_index As Long, ByVal buff
As String, ByVal max_length As Long, ByVal buff As String) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'atom_index' - index for the row of array, 'molecule_index' - index for the column of the array, 'string' pointer to the*

new, NULL-terminated string.

FORTRAN: *'hsv' - the HSV code, 'atom_index' - index for the row of array, 'mol_index' - index for the column of the array, 'buff' - character array containing Fortran string.*

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

The both atom and molecule indices start with one.

See Also

hcGetStrArrElm, hcGetStr, hcSetStr

hcSetBlock

The function updates the contents of the whole HSV variable irrespevtively of its type.

API header

```
int hcSetBlock(int var, char* buff, int length);
```

FORTRAN interface

```
logical function hfSetBlock( hsv, buff, res_length)
integer*1 buff
```

VISUAL BASIC declaration

```
Declare Function hbSetBlock Lib "hapi.dll" Alias "hcSetBlock" (ByVal var As
Long, ByVal buff As String, ByVal max_length) As Long
```

Parameters

C/C++: *'var' - the HSV code, 'buff' pointer to the buffer where the data is copied, 'length' - the size of 'buff' (in bytes)*

FORTRAN: *'hsv' - the HSV code, 'buff' - integer*1 (byte) array containing the data,*

'res_length' - the size of 'buff' (in bytes)

VB: *N/A*

Return Value

The function returns 1 if the operation was successful and 0 otherwise.

Remarks

Some HSV variables, particularly, some vectors and arrays have the type of element which is not scalar. The hcGetBlock was provided to get access to that kind of variable. However, the user's application is responsible for the interpretation of the data in the block and proper setting all of the individual elements of the block.

See Also

hcGetBlock, hcQueryBin

Functions for Processing Notifications

hcNotifyStart

The function requests for notifications about any change of the HSV variable.

API header

```
int hcNotifyStart(LPSTR var_name);
```

FORTRAN interface

```
logical function hfNotifyStart(hsv_name)
character*(*) hsv_name
```

VISUAL BASIC declaration

```
Declare Function hbNotifyStart Lib "hapi.dll" Alias "hcNotifyStart" (ByVal
var_name As String) As Long
```

Parameters

C/C++:	*'var_name' - HSV name (text) for which notification is requested.*
FORTRAN:	'hsv_name' - *HSV name (text) for which notification is requested.*
VB:	*N/A*

Return Value

The function returns 1 if the notification request is accepted.

Remarks

The function simply request for notification. H how the notification will be processed is specified by the hcNotifySetup function.

See Also

hcNotifyStop, hcNotifySetup

hcNotifyStop

The function cancels the request for notifications about the change of the HSV variable.

API header

```
int hcNotifyStop(LPSTR var_name);
```

FORTRAN interface

```
logical function hfNotifyStop( hsv_name)
character*(*) hsv_name
```

VISUAL BASIC declaration

```
Declare Function hbNotifyStop Lib "hapi.dll" Alias "hcNotifyStop" (ByVal
var_name As String) As Long
```

Parameters

C/C++:	*'var_name' - HSV name (text) for which notification request is cancelled.*

FORTRAN: *'hsv_name' - HSV name (text) for which notification request is cancelled*

VB: *N/A*

Return Value

The function returns 1 if the notification request was canceled.

Remarks

The function stops the notification irrespective of the method for notification processing.

See Also

hcNotifyStart, hcNotifySetup

hcNotifySetup

The function establishes how the notifications have to be processed.

API header

```
int hcNotifySetup(PFNB pCallBack,int NotifyWithText);
```

FORTRAN interface

```
logical function hfNotifySetup (FnCallback, TextAdviseFlag)
logical TextAdviseFlag
```

VISUAL BASIC declaration

```
Declare Function hbNotifySetup Lib "hapi.dll" Alias "hcNotifySetup" (ByVal clb
As Long, ByVal NotifyWithText As Long) As Long
```

Parameters

C/C++: *'pCallBack' - pointer to the callback function designed to process notifications. However, the user may provide NULL parameter for pCallBack and in this case the Notification Agent will be used. The last parameter '`NotifyWithText`' orders text notifications if is 1 and binary if it is 0.*

FORTRAN: *'FnCallBack' - pointer to the callback function designed to process notifications. However, the user may provide 0 as a parameter for FnCallBack and in this case the Notification Agent will be used. The last parameter 'TextAdviseFlag' orders text notifications if is 1 and binary if it is 0.*

VB: *N/A*

Return Value

The function returns 1 if the notification request was cancelled.

Remarks

The user application may define its own function to process notification in one of the forms:

```
typedef VOID (*PFNB)(int var, char* data, int length);
```
for the binary notifications, or:

```
typedef VOID (*PFNX)(char* name, char* data);
```
for the text notifications.

In both cases the notification agent will not be used and the application is responsible for processing, storing or buffering the incoming data through the callback function. This is the best method for processing notifications.

However, as it was noted in Chapter 11, many types of applications cannot receive or properly process notifications. This include all console based applications, FORTRAN programs, external Tcl/Tk programs etc. Providing NULL as the callback address parameter automatically starts the Notification Agent (0 in Fortran).

See Also

hcNotifyStart, hcNotifyStop

hcNotifyDataAvail

The functions checks if the Notification Agent has any not-processed notifications in its buffers.

API header

```
int hcNotifyDataAvail(void);
```

FORTRAN interface

```
integer function hfNotifyDataAvail()
```

VISUAL BASIC declaration

```
Declare Function hbNotifyDataAvail Lib "hapi.dll" Alias "hcNotifyDataAvail"
(ByVal var_name As String) As Long
```

Parameters

The function has no parameters

Return Value

The function returns 1 if there is any unprocessed notification or 0 otherwise.

Remarks

The application may call the function as often as required; a call to the function deschedules the time slicing of the Windows operating system, so the application does not consume much processing time.

See Also

hcNotifyStart, hcNotifySetup

hcGetNotifyData

The functions gets data arriving from a notification previously checked.

API header

```
int hcGetNotifyData(char* name,char *buffer, DWORD MaxBuffLength);
```

FORTRAN interface

```
integer function hfGetNotifyData( name. result, res_length)
integer res_length
character*(*) name, result
```

VISUAL BASIC declaration

```
Declare Function hbGetNotifyData Lib "hapi.dll" Alias "hcGetNotifyData" (ByVal
name As String, ByVal buffer As String, ByVal MaxBuffLength As Long) As Long
```

Parameters

C/C++: *'name'- is the address of the buffer where the name of the variable is placed. 'buffer' - is the address of the buffer where incoming data will be copied, 'MaxBufferLength' is the maximum size of the data block that 'buffer' can accept.*

FORTRAN: *'name'- is the character array where the name of the variable is placed. 'result' - is the character array where incoming data will be copied, 'res_length' is the maximum size of the data block that 'result' can accept (in bytes).*

VB: *N/A*

Return Value

The function returns the number of bytes copied by the function to the buffer 'buffer'.

Remarks

Each call to hcGetNotifyData copies the notification message to the provided buffers and discards that message, deleting the appropriate buffers in the Notification Agent area. This means that the user's application listening for notifications must properly process all discarded (and copied) messages.

See Also

hcNotifyStart, hcNotifySetup, hcNotifyDataAvail

Functions For Memory Allocation

hcAlloc

The function allocates a memory block.

API header

```
void* hcAlloc(size_t n_bytes);
```

FORTRAN interface

```
This function is unavailable for Fortran programs
```

VISUAL BASIC declaration

```
This function is unavailable for Visual Basic programs
```

Parameters

C/C++:	*'n_bytes' number of bytes to allocate.*
FORTRAN:	N/A
VB:	*N/A*

Return Value

The function returns a pointer to the allocated block, or NULL if the allocation was not successful.

Remarks

The 'hcAlloc' allocates memory for both internal HAPI needs and possible user requirements. However, the user may use regular C/C++ allocation routines.

See Also

hcFree

hcFree

The function deallocates a block of memory previously allocated by a call to hcAlloc.

API header

```
void hcFree(void* pointer);
```

FORTRAN interface

```
This function is unavailable for Fortran programs
```

VISUAL BASIC declaration

```
This function is unavailable for Visual Basic programs
```

Parameters

C/C++:	*'pointer' is the pointer obtained by a previous call to hcAlloc.*
FORTRAN:	N/A
VB:	*N/A*

Return Value

The function does not return any data.

Remarks

The main use for hcFree is after processing the data returned by hcQueryTxt and hcQueryBin. See description for these functions.

See Also

hcAlloc, hcQueryBin, hcQueryTxt

Auxiliary Functions

hcShowMessage

The function displays message box with provided string.

API header

```
void hcShowMessage(LPSTR message);
```

FORTRAN interface

```
subroutine hfShowMessage( str)
character*(*) str
```

VISUAL BASIC declaration

```
This function is useless for Visual Basic programs
```

Parameters

C/C++: *'message' points to NULL terminated string containing message to be displayed.*

FORTRAN: 'str' - Fortran string containing the message to display.

VB: *N/A*

Return Value

The function does not return any data.

Remarks

The function may be useful for debuging programs that cannot easily display regular Windows messages (like most console-based programs).

See Also

hcSetTimeouts

The function sets new timeout values for execution, querying and other types of communication.

API header

```
void hcSetTimeouts(int ExcTimeOut,int QryTimeOut,int RstTimeOut);
```

FORTRAN interface

```
subroutine hfSetTimeouts(t_exc, t_qry, t_other)
integer t_exc, t_qry, t_other
```

VISUAL BASIC declaration

```
Declare Function hbSetTimeouts Lib "hapi.dll" Alias "hcSetTimeouts" (ByVal
ExcTimeOut As Long, ByVal QryTimeOut As Long, ByVal RstTimeOut As Long) As Long
```

Parameters

C/C++: *'ExcTimeOut' is the new time-out value (in miliseconds) for the execution of commands sent to HyperChem, 'QryTimeOut' is the new time-out value (in miliseconds) for the processing requests for HSV variables ,'RstTimeOut' is the new time-out value for processing controling commands, like notification requests etc.*

FORTRAN: 't-exc' *is the new time-out value (in milliseconds) for the execution of commands sent to HyperChem, 't_qry' is the new time-out value (in milliseconds) for processing requests for HSV variables ,'t_other' is the new time-out value for process controlling commands, like notification requests, etc.*

VB: *N/A*

Return Value

The function does not return any data.

Remarks

The default value for all time-outs is about 65 seconds (0xFFF0.) After this time expires and any command, query or other operation has not finished, the error condition is invoked. The user may increase the time-out value for any

of these three types. The most common situation where the time expires is associated with execution of commands controlled by ‘’ExcTimeOut’ and t_exc’.

See Also

hcLastError

This function retrieves code and messages associated with the last error associated with an HAPI operation.

API header

```
int hcLastError(char* LastErr);
```

FORTRAN interface

```
integer function hfLastError( error)
character*(*) error
```

VISUAL BASIC declaration

```
Declare Function hbLastError Lib "hapi.dll" Alias "hcLastError" (ByVal
last_error As String) As Long
```

Parameters

C/C++: *The ‘LastErr’ pointer to string that will receive the text message associated with the last error. The string should be of ‘hcMaxMessSize’, with ‘hcMaxMessSize’ defined in ‘hc.h’*

FORTRAN: ‘error’ is the Fortran string that receives the last error message.

VB: *N/A*

Return Value

The function returns a value indicating how severe the error was. There are three possibilities:
errNO_ERROR - last operation was completed successfully

errNON_FATAL - last operation has not completed successfully, but the

program can continue.

errFATAL - last operation caused a severe error and the application cannot continue.

The flags errNO_ERROR, errNON_FATAL and errFATAL are defined in 'hc.h'.

Remarks

The hcLastError function is most useful when the user setsthe error action flag to errACTION_NO using hcSetErrorAction. In this case the error does not invoke messages on the screen asking for user intervention.

See Also

hcSetErrorAction, hcGetErrorAction

hcGetErrorAction

This function retrives the flag informing you how HAPI processes errors.

API header

```
int hcGetErrorAction(void);
```

FORTRAN interface

```
integer function hfGetErrorAction()
```

VISUAL BASIC declaration

```
Declare Function hbGetErrorAction Lib "hapi.dll" Alias "hcGetErrorAction" ()
As Long
```

Parameters

C/C++: *The function takes no parameters.*

FORTRAN:

VB:

Return Value

The function returns the flag that may be a sum of the following flags defined in 'hc.h':
errACTION_NO - do not perform any action on any error

errACTION_MESS_BOX - display message box with error message

errACTION_DISCONNECT - disconnect the application from HyperChem

errACTION_EXIT- immediately exit from application

errDDE_REP - report low level DDE error messages

errDDE_NO_REP - do not report low level DDE error messages

Remarks

The function should be called before the user changes the error processing method by a call to hcSetErrorAction, and its value stored for later use in restoring the original error processing method.

See Also

hcLastError, hcSetErrorAction

hcSetErrorAction

This function changes the way errors are processed.

API header

```
void hcSetErrorAction(int err);
```

FORTRAN interface

```
subroutine hfSetErrorAction(action)
integer action
```

VISUAL BASIC declaration

```
Declare Function hbSetErrorAction Lib "hapi.dll" Alias "hcSetErrorAction"
(ByVal action As Integer) As Long
```

Parameters

C/C++: *'err' - the value indicating how errors are to be processed. Must be a sum of the following flags defined in 'hc.h':*
errACTION_NO - do not perform any action on any error
errACTION_MESS_BOX - display message box with error message
errACTION_DISCONNECT - disconnect the application from HyperChem
errACTION_EXIT - immediately exit from application
errDDE_REP - report low level DDE error messages
errDDE_NO_REP - do not report low level DDE error messages

FORTRAN: 'action' -indicates how errors are processed. See above for symbolic names.

VB: *N/A*

Return Value

The function does not return any data.

Remarks

There are situations when the user does want to change the default error processing. By combining the value of flags in the appropriate sum it is possible to get different actions on errors, ranging from no action (errACTION_NO) to full information (errACTION_MESS_BOX | errDDE_REP) and, possibly, exiting from current application (errACTION_EXIT).

See Also

hcGetErrorAction, hcLastError

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T