The Geochemist’s Workbench®

Release 12

GWB Reference Manual

The Geochemist's Workbench®

Release 12

GWB Reference Manual

Craig M. Bethke
Brian Farrell
Sharon Yeakel

Aqueous Solutions, LLC
Champaign, Illinois

Printed October 15, 2018

The GWB software was originally developed by the students, staff, and faculty of the Hydrogeology Program in the Department of Geology at the University of Illinois Urbana-Champaign. The package is currently developed and maintained by Aqueous Solutions LLC at the University of Illinois Research Park.

Address inquiries to

Aqueous Solutions LLC
301 North Neil Street, Suite 400
Champaign, IL 61820 USA

Cover photo: Salinas de Janubio by Jorg Hackemann.

Contents

Chapter List

Contents

Introduction

This **GWB Reference Manual** contains information about the command line interface on the **Command** pane, the format of thermodynamic datasets, the report command, control scripts, the plug-in feature for running the GWB applications from within your own applications, unit conversion within the programs, and manipulating graphics output.

The manual also describes several legacy features: text-format scatter data, using scripts to process multiple analyses, and the remote control feature. Each of these has been superseded by the **GSS** application or the plug-in feature.

This manual is intended as a supplement to the GWB manuals: **GWB Essentials Guide**, **GWB Reaction Modeling Guide**, **GWB Reactive Transport Modeling Guide**, and the **GWB Command Reference**.

Please consult the latter manual for specifics about the commands used to configure the GWB programs.

1.1 Chapters in the manual

This manual contains chapters that provide details about specific features of the GWB software package:

- **Command Line Interface** — The features of the user interface for the **Command** pane, including spelling completion, history substitution, and the built-in calculator.
- **Thermo Datasets** — Information about the formatting and content of the thermodynamic databases the GWB programs can read. This information is useful if you need to modify the database, or create your own.
- **Report Command** — Format and use of the “report” command, which returns the results of calculations. This command provides a means of transmitting results to control scripts and to programs running a GWB application as a plug-in or by remote control.
- **Control Scripts** — How to set up within GWB input file scripts containing loops, branches, if checks, and so on.

- **Plug-in Feature** — Details how to use the capabilities of the GWB applications through the functions of a DLL.
- **Units Recognized** — A complete table of the unit names to be used in the commands.
- **Graphics Output** — How you can manipulate the graphical output from **Act2**, **Tact**, **Gtplot**, **P2plot**, and **Xtplot**.

1.2 Legacy features

The manual also desribes several legacy features of the software:

- **Scatter Data** — The legacy method of adding scatter data to a diagram by importing a specially formatted table from a text file. The preferred method is to use a **GSS** spreadsheet as described in the **GWB Essentials Guide**.
- **Multiple Analyses** — Examples of how to process a number of chemical analyses from a spreadsheet and save the results to the spreadsheet.
- **Remote Control** — Details the deprecated legacy method of how you can run the GWB applications as slave processes from other programs and software environments. This method has been replaced by the **Plug-in Feature**.

Command Line Interface

The command line interface for **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** includes a number of special features that you will find increasingly helpful as you gain experience.

2.1 Spelling completion

Rxn, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** can complete the spelling of chemical and mineral names. The feature also completes the names of program commands and dataset names. To complete a spelling, begin typing the name and touch the tab (or, on most computers, escape) key. For example, if you type

```
1 free mole Musc[tab]
```

the program will complete the line

```
1 free mole Muscovite
```

When the program cannot identify a unique name from the letters provided, it will cycle through the possible completions with subsequent tab key presses. To list the possible completions, you can type **Ctrl+D**. For example, if you type

```
swap Al2^D
```

the program will respond with

```
    Al2(OH)2++++     Al2(SO4)3       Al2(SO4)3:6H2O
swap Al2|
```

leaving the cursor (“|”) in position to continue the command.

2.2 History substitution

Rxn, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** maintain history lists. Previously executed commands are stored in the user's profile directory (e.g., "c:\Documents and Settings\ jones\Application Data\GWB") in datasets such as "rxn_history.dat". If you type the command

```
history 10
```

the program lists the previous ten commands executed.

Each of the history substitution functions of the C-shell are available within the programs. For example, typing

```
!!
!10
!swap
!?HCO3
```

causes the program to re-execute, respectively, the previous command, command number 10, the last command that began with "swap", and the last command that contained the string "HCO3".

Entries of the form

```
^string1^string2
```

replace the occurrence of "string1" with "string2" in the previous command, so that you can avoid retyping lengthy commands after simple errors. Finally, typing

```
!10-15
```

causes the program to re-execute commands 10 through 15. This latter feature is an extension to the C-shell protocol.

2.3 Special characters

The following special characters are used in the **Command** pane in **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t**:

Ctrl+D	Show choices for spelling completion of chemical names, commands, or file names.
Ctrl+F	Clear screen.
Ctrl+H	Backspace over a character.
Ctrl+N	Scroll forward through your command history to give the next command in list.
Ctrl+P	Scroll backward through your command history to give the previous command in list.
Ctrl+U	Backspace over entire line of input.
Ctrl+W	Backspace over previous word of input.
`Tab` *or* `Esc`	Cycle through choices for spelling completion of chemical names, commands, or file names.
`\`	Continue a command from one line to the next.

2.4 Calculator

Programs **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** provide an online calculator with these functions:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponentiation
`( )`	Grouping of terms
`ln`	Natural logarithm
`log`	Common logarithm
`abs`	Absolute value
`sqrt`	Square root
`exp`	Exponentiation of e
`sin, cos, tan, cot, sec, csc`	Trigonometric functions (arguments in radians)
`e`	Value of e
`pi`	Value of π

The online calculator is especially useful to a geochemist for converting numbers to their logarithms and vice versa, but can be used to evaluate any numerical expression. To use the calculator, type an expression at the prompt. Examples:

```
log .0003
10^-4.5
(200 - 32) * 5/9
```

In each case, the program evaluates the expression and returns its numerical value.

2.5 Startup files

Upon startup, **Rxn** looks for a file such as “rxn_startup.rxn” in a user’s profile directory (found by typing %appdata% in the Windows Explorer Address bar, e.g., “c:\Documents and Settings\jones\Application Data\GWB”) and executes any commands in it; the other modeling programs similarly look for files named “act2_startup.ac2”, “spece8_startup.sp8”, and so on. These datasets provide a means for you to customize the working environment of each program. An “act2_startup.ac2” file including the commands

```
background grey
font Times
```

for example, will cause **Act2** to produce plots with grey backgrounds and Times lettering, unless told otherwise.

2.6 On-line documentation

You can obtain on-line help for any of the programs on the **Docs** pane of the GWB dashboard, or using the “Help” pulldown menu on the menubar of any GWB program. The entire manual set, including this User’s Guide, can be accessed from the “Help” pulldown.

2.7 System commands

You can execute (“fork”) DOS commands from the command lines of **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t.** To do so, type a “$” followed by the desired DOS command. Example:

```
$print React_output.txt
```

When a system command is executed, a “Command Prompt” window will appear briefly on your screen. Due to limitations of the Windows operating system, you cannot fork a command that requires user input, and you will not be able to see any output (including error messages) that might be generated by the command.

2.8 Text size in the GWB windows

You can control the font and size of the text within the program shells for **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** by choosing the desired font and point size from **View** → **Appearance. . . .** "Reset" will change the point size back to the default value.

2.9 Keyboard shortcuts

Ctrl+Break	Break program (used during program execution, it stops the calculation and returns you to the command prompt)
Ctrl+A	Select all
Ctrl+C	Copy
Ctrl+Shift+C	Load conductivity data (SpecE8, React, Phase2, X1t, X2t)
Ctrl+F	Clear screen
Ctrl+G	Go, run calculation of the model
Ctrl+Shift+G	Load scatter data (Act2, Tact)
Ctrl+Shift+G	Go single, run on a single processor (X1t, X2t)
Ctrl+I	Go initial, calculate the initial state of the medium (React, Phase2, X1t, X2t)
Ctrl+Shift+L	Launch Gtplot, P2plot, or Xtplot to show results (SpecE8, React, Phase2, X1t, X2t)
Ctrl+M	Save image... (Act2, Tact)
Ctrl+N	Add an entry to the basis
Ctrl+O	Read script (open data file)
Ctrl+Q	Quit the program
Ctrl+R	Reset configuration
Ctrl+Shift+R	Resume, restore the configuration from when the program was last exited
Ctrl+S	Save As...
Ctrl+Shift+S	Load sorbing surfaces (Rxn, SpecE8, React, Phase2, X1t, X2t)
Alt+S	Reset the size of the window to the default
Ctrl+T	Load thermo dataset
Ctrl+Shift+T	Load reaction trace (Act2, Tact)
Ctrl+U	Update trace (Act2, Tact)
Ctrl+V	Paste
Ctrl+Shift+W	Change working directory
Ctrl+X	Cut
Ctrl+Y	Redo (GSS)
Ctrl+Z	Undo (GSS)
F1	GWB Reference Manual

Thermo Datasets

The databases of thermodynamic data used by the programs are ascii (or character) files that you edit with TEdit, the thermo editor supplied as part of the GWB software. You can alternatively change a thermo file using a text editor, such as “Notepad” under MS Windows.

You are free to alter existing databases such as “thermo.tdat” by changing data or adding species, minerals, and so on. When changing a database, it is a good idea to copy the original database to a file with a new name, and then alter that file. You can also create your own databases by following the dataset format.

You access the new file from the GWB apps with the “read” command, or by opening the **File** →**Open** →**Thermo Data. . .** dialog. You can set a specific dataset to be read by default when one of the GWB apps starts. To do so, set the dataset as the default thermodynamic file in **File** →**Preferences. . .** (see **Thermodynamic datasets** in the **Introduction** to the **GWB Essentials Guide**).

The information in this chapter applies to the “jul17” format, which is used beginning with the GWB12 release. A description of legacy datasets appears at the end of the chapter.

3.1 Thermodynamic dataset format

Each thermo dataset is composed of the following sections:

1. Header lines, which identify the dataset
2. Data tables
3. Elements included in the dataset
4. Basis species
5. Redox couples
6. Aqueous species
7. Free electron
8. Minerals
9. Gases

10. Oxide components

11. Virial coefficients, for datasets invoking a virial (“Pitzer”) activity model

Sections 3–10 begin with a header line such as

```
46 elements
```

which identifies the number of elements, species, and so on in each section. The count is ignored in the current software, but very old GWB releases require it to be accurate. A line

```
-end-
```

marks the end of each section.

You can include comment lines, identified by a “*” as the first character, freely within the dataset. The programs read the data word-by-word, so it is not necessary to count spaces or align columns when adding new data.

3.1.1 Header lines

A group of header lines appears at the top of the dataset:

```
dataset of thermodynamic data for gwb programs
dataset format: jul17
activity model: debye-huckel
fugacity model: tsonopoulos
```

These lines identify the dataset, its format, and the activity model to be invoked. The current format is “jul17”; earlier formats are described in the last section of this chapter.

The activity model should be “debye-huckel”, “h-m-w”, “phrqpitz” (currently equivalent to “h-m-w”), “phreeqc”, “wateq4f”, or “minteq”. The “pitzer” activity model is outmoded and no longer supported. The fugacity model used to calculate gas partial pressure may be “tsonopoulos”, “peng-robinson”, or “spycher-reed”.

3.1.2 Data tables

The following section contains tables showing how various coefficients vary with temperature. The first table contains eight principal temperatures at which the data tables and log K’s for species and so on are tabulated.

```
* temperatures
        0.0000     25.0000     60.0000    100.0000
      150.0000    200.0000    250.0000    300.0000
```

Most commonly, the principal temperatures are 0°C, 25°C, 60°C, 100°C, 150°C, 200°C, 250°C, and 300°C, but you can choose other values.

In isothermal runs in which temperature is set to one of the principal temperatures, the GWB applications take the corresponding value from each data table. In a run at 25°C, given the principal temperatures above, for example, the application takes the second entry in each table in the header section, and the second log K entry for each chemical species in the dataset. Where temperature differs from a principal temperature, the application fits each table to a polynomial

$$v = a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4$$

with respect to temperature T, in °C. You can use Rxn to quickly see the polynomial fit of the log K for any chemical species:

```
react Quartz
long
go
```

for example.

Following the temperature table are tables giving pressure, coefficients for calculating parameters in activity coefficient correlations, and so on, at each of the principal temperatures. The tables look like:

```
* pressures
        1.0134     1.0134     1.0134     1.0134
        4.7600    15.5490    39.7760    85.9270
* debye huckel a (adh)
        .4913      .5092      .5450      .5998
        .6898      .8099      .9785     1.2555
...
```

The parameters in these tables are

- Pressure, in bar,
- The Debye-Hückel parameters A, B, and $\dot{B}$,
- Coefficients for calculating the activity coefficients for CO_2 and some other electrically neutral species, and
- Coefficients for calculating the activity of water.

3.1.3 Elements

The next section contains the elements of which species and so on in the database are composed, and each element's chemical symbol and mole weight. The entries in this section look like

```
Oxygen      (O )    mole wt.=     15.9994
Silver      (Ag)    mole wt.=    107.8680
Aluminum    (Al)    mole wt.=     26.9815
Americium   (Am)    mole wt.=    241.0600
...
```

3.1.4 Basis species

The basis species for the dataset, beginning with water (H_2O), are defined in the next section. The entry for each species contains its charge, ion size parameter in angstroms (for calculating its activity coefficient), mole weight (g/mol), and elemental composition

```
H2O
     charge=   0.0    ion size=  0.0 A    mole wt.=   18.0152
     2 elements in species
      1.000 O              2.000 H
Ag+
     charge=   1.0    ion size=  2.5 A    mole wt.=  107.8680
     1 elements in species
      1.000 Ag
Al+++
     charge=   3.0    ion size=  9.0 A    mole wt.=   26.9815
     1 elements in species
      1.000 Al
Am+++
     charge=   3.0    ion size=  9.0 A    mole wt.=  241.0600
     1 elements in species
      1.000 Am
...
```

The ion size parameter (a_o) has special meaning for neutrally charged aqueous species in the thermo dataset. For neutral species with $a_\text{o} \geq 0$, the species' activity coefficient is set to one. When $a_\text{o} = -1/2$, the activity coefficient is calculated from the "CO2" coefficients in the data table section, according to equation 8.6 in the "Geochemical and Biogeochemical Reaction Modeling" text. When $a_\text{o} \leq -1$, the logarithm of the activity coefficient is set to the product $\dot{B} \times I$, where $\dot{B}$ is given by the data tables above, and I is true ionic strength.

Whenever a database is to include protonation and deprotonation reactions, the list of basis entries needs to include the hydrogen ion, H^+, labeled as "H+". Databases treating redox require as part of the list either dissolved dioxygen, $\mathrm{O_2(aq)}$, or dissolved dihydrogen $H_2(aq)$; the species must be labeled, respectively, “O2(aq)” and “H2(aq)”. The species you choose, $O_2(aq)$ or $H_2(aq)$, is the database’s “redox pivot”.

3.1.5 Redox couples

Redox coupling reactions for the dataset are found in the following section. The Fe^{+++}/Fe^{++} couple, for example, is represented

```
Fe+++
     charge=  3.0      ion size=  9.0 A    mole wt.= 55.8470
     4 species in reaction
      -0.500        H2O        1.000 Fe++             1.000 H+
       0.250 O2(aq)
       -10.0553    -8.4878    -6.6954    -5.0568
        -3.4154    -2.0747    -0.8908     0.2679
```

The first two lines identify the redox species and give its charge, ion size parameter, and mole weight.

The subsequent lines show the reaction by which the redox species dissociates. This reaction can be written in terms of any of the basis species in the previous section of the dataset, or any redox species that have been defined to this point in the current section. You may also use gaseous dioxygen or dihydrogen, or the free electron, as described in the next paragraph. Note that you set reactions in the dataset 3 species per line, until the reaction is complete. The final lines give log K values for this reaction at each of the principal temperatures.

The manner in which you represent electron acceptance and donation in the coupling reactions depends on the redox pivot you have chosen. Where the pivot is $O_2(aq)$, you may balance redox reactions in terms of aqueous or gaseous dioxygen, “O2(aq)” or “O2(g)”:

```
Fe+++
     charge=  3.0       ion size=  9.0 A       mole wt.=   55.8470 g
     4 species in reaction
      -0.500 H2O              1.000 Fe++              1.000 H+
       0.250 O2(g)
        -9.3901    -7.7630    -5.9309    -4.2756
        -2.6496    -1.3462    -0.2258     0.8704
```

Or, you may set a half-cell reaction in terms of the free electron, “e-”

```
Fe+++
     charge=  3.0      ion size=  9.0 A      mole wt.=   55.8470 g
     2 species in reaction
       1.000 Fe++           -1.000 e-
        13.3713   13.0127   12.5821   12.1903
        11.8236   11.5751   11.4559   11.5415
```

Similarly, where the pivot is $H_2(aq)$, you may use “H2(aq)”, “H2(g)”,

```
Fe+++
     charge=  3.0      ion size=  9.0 A      mole wt.=   55.8470 g
     3 species in reaction
       1.000 Fe++            1.000 H+             -0.500 H2(g)
        13.6804   13.0127   12.2194   11.4873
        10.7750   10.2446    9.8895    9.7741
```

or “e-”.

In datasets that do not include a redox pivot within the basis, you would normally not include redox coupling reactions. In that case, the section would look like

```
  0 redox couples
  -end-
```

3.1.6 Aqueous species

The next section contains the aqueous species to be considered in addition to the basis and redox species. The entry for $CaCl^+$, for example, is

```
CaCl+
     charge= 1.0      ion size= 4.0 A      mole wt.= 75.5330
     2 species in reaction
       1.000 Ca++           1.000 Cl-
        -0.9687   -0.7000  -0.5157    -0.4688
        -0.5789   -0.8602  -1.3560    -2.2451
```

The entry contains the reaction for dissociating the species to the basis and redox species, and the log K values for this reaction.

To maintain the software’s ability to couple and decouple redox reactions, you should balance reactions in this and following sections by avoiding the use, wherever possible, of $O_2(aq)$, $O_2(g)$, $H_2(aq)$, $H_2(g)$, and e^-. To do so, you balance the reactions in terms of species of the same oxidation state as the species in question. The entry for $H_2S(aq)$, for example,

```
H2S(aq)
     charge=   0.0      ion size=  4.0       A mole wt.=
34.0758 g
     2 species in reaction
       1.000 H+              1.000 HS-
         -7.6500  -6.9500  -6.6800   -6.6100
         -6.7900  -7.1700  -7.7200   -8.4300
```

is properly balanced in terms of the redox species HS^-, rather than the basis species SO_4^{--}.

Note that for any basis species, redox couple, or aqueous species, you may specify a stoichiometric formula

```
Acetic acid                         formula= HCH3COO
     charge=  0       ion size=  4.0 A       mole wt.=   60.0524 g
     2 species in reaction
        1.000 H+              1.000 CH3COO-
        -4.7743      -4.7563      -4.8079      -4.9640
        -5.3017      -5.8241     500.0000     500.0000
```

by appending a “formula=” field to the species’ name.

3.1.7 Free electron

The half-cell reaction representing take-up of the free electron is written in a form that reflects the database’s redox pivot. Where the pivot is $O_2(aq)$, the reaction is written in terms of either “O2(aq)” or “O2(g)”:

```
e-
     charge= -1.0       ion size=  0.0 A       mole wt.=    0.0000 g
     3 species in reaction
       0.500 H2O             -0.250 O2(g)           -1.000 H+
        22.7614   20.7757   18.5130   16.4658
        14.4732   12.9213   11.6817   10.6711
```

For $H_2(aq)$, the reaction is given in terms of “H2(aq)” or “H2(g)”:

```
e-
     charge= -1.0       ion size=  0.0 A       mole wt.=    0.0000 g
     2 species in reaction
       0.500 H2(g)           -1.000 H+
        -0.3091     0.0000     0.3627     0.7030
         1.0486     1.3305     1.5663     1.7673
```

3.1.8 Minerals

Minerals to be included in the database are found in the next section. The entry for the sodium feldspar albite looks like

```
Albite                          type= feldspar
     formula= NaAlSi3O8
     mole vol.=  100.250 cc      mole wt.= 262.2230 g
     5 species in reaction
       2.000 H2O            1.000 Na+            1.000 Al+++
       3.000 SiO2(aq)      -4.000 H+
         3.9160     3.0973    1.9915     0.9454
        -0.0499    -0.8183   -1.5319    -2.5197
```

The initial lines in the entry give the mineral's name, type, formula, and molar volume (cm^3/mol) and weight (g/mol). The remaining lines give the reaction by which the mineral decomposes to basis and redox species and the corresponding log K values at the principal temperatures. As with the aqueous species, the reaction should be written without change in oxidation state, if possible.

3.1.9 Gases

The next section contains the gases considered. The entry for CO_2(g), for example, is

```
CO2(g)
     mole wt.=    44.0098 g
     chi=       -1430.87      3.598   -.00227376     3.47644   -.0104247   8.46271e-6
     Pcrit=      73.8 bar       Tcrit=      304.1 K       omega=         .239
     3 species in reaction
      -1.000 H2O              1.000 H+                  1.000 HCO3-
        -7.6827    -7.8184    -8.0628    -8.3849
        -8.8297    -9.3208    -9.8841   -10.6132
```

The first lines give the name and mole weight (g/mol) of the gas.

The next two lines, which are optional and can appear in either order, hold data for calculating fugacity coefficients. A line starting with "chi=" gives the factors a through f used in the Spycher-Reed method to calculate the fugacity coefficient φ

$$\ln\varphi = \left(\frac{a}{T_K^2} + \frac{b}{T_K} + c\right) P + \left(\frac{d}{T_K^2} + \frac{e}{T_K} + f\right)\frac{P^2}{2} \tag{3.1}$$

as a function of absolute temperature T_K, and pressure P, in bars.

A line beginning "Pcrit=" gives the data needed to evaluate the Tsonopoulos and Peng-Robinson pressure models: the critical point pressure P_{cr} and temperature T_{cr}, and the acentric factor ω. For polar or hydrogen bonding gases, like H_2O(g), the line may be extended to include factors a and b

```
     Pcrit=   221.2 bar   Tcrit=   647.3 K   omega=   .344   a= -.0109   b=  0.0
```

used for such gases by the Tsonopoulos model.

As before, the remaining lines give the gas' dissociation reaction and corresponding log K values.

3.1.10 Oxide components

The oxide components are used as reactants in simulations that, for example, model the dissolution of a glass phase. An example is

```
Al2O3
     mole wt.=  101.9616 g
     3 species in reaction
      -6.000 H+              2.000 Al+++              3.000 H2O
```

Note that since components are fictive entities used to describe bulk composition, they have no thermodynamic stability and hence there are no entries for log K values.

3.1.11 Virial coefficients

Datasets that invoke the virial activity model include a final section containing the virial coefficients and any data for describing variation of the coefficients with temperature. The coefficients $\beta^{(0)}$, $\beta^{(1)}$, $\beta^{(2)}$, c^{ϕ}, θ, λ, and ψ can vary with temperature; they vary in value according to the polynomial

$$\beta = \beta_{25} + c_1 (T_K - T_r) + c_2 \left(\frac{1}{T_K} - \frac{1}{T_r} \right) + c_3 \ln \left(\frac{T_K}{T_r} \right) + c_4 \left(T_K^2 - T_r^2 \right) + c_5 \left(\frac{1}{T_K^2} - \frac{1}{T_r^2} \right)$$

Here, β represents the virial coefficient in question and β_{25} is its value at 25°C, T_K is absolute temperature, and T_r is 298.15 K.

Each coefficient is defined on a line containing at a minimum the value at 25°C and optionally one or more of the polynomial coefficients $c_1 - c_5$. Any omitted entries for the polynomial are treated as zero values. Hence, a line containing β_{25} and c_1 defines the virial coefficient in terms of its 25°C value and its first temperature derivative.

There are four segments of virial coefficients in the section:

- Values of $\beta^{(0)}$, $\beta^{(1)}$, $\beta^{(2)}$, c^{ϕ}, $\alpha^{(1)}$, and $\alpha^{(2)}$ for cation-anion pairs,
- Values of θ for anion-anion pairs,
- Values of λ for ion-neutral species pairs, and
- Values of ψ for species triplets.

An example block for the cation-anion pair Na^+-Cl^- is

```
Na+        Cl-
   beta0   = 0.0765    0.008946    -777.03    -4.4706    -3.3158E-6
   beta1   = 0.2664    6.1608E-5   0.0        0.0        1.0715E-6
   beta2   = 0.0
   cphi    = 0.00127   -4.655E-5   33.317     0.09421
   alpha   = 2.0
   alpha2  = 0.0
```

In this case, the value of $\beta^{(0)}$ at 25°C is 0.0765, and the value's temperature variation is defined by polynomial coefficients c_1–c_4 of $0.008946, -777.03, -4.4706$, and -3.3158×10^{-6}.

3.2 Legacy formats

Three legacy formats for the thermo databases may be used with current releases of the software. The legacy formats are labeled "oct13", "oct94" and "feb94". The first two formats are preferred for "h-m-w" and "phrqpitz" activity models, and latter accepts the outmoded "pitzer" model.

GWB releases 11.0 and earlier work only with these formats; they do not accept databases in the current "jul17" format. GWB 9.0 and earlier, likewise, do not recognize "oct13" format databases.

The differences between the legacy and current formats are summarized below.

3.2.1 Header lines

Datasets predating the "jul17" format do not support partial pressure calculations and do not contain a "fugacity model" header line.

3.2.2 Tables

The datasets in the "oct94" and "feb94" formats include four additional tables:

- One table containing the log K values for the half-cell reaction

$$2\ H_2O \rightleftarrows O_2(g) + 4\ H^+ + 4\ e^-$$

  and

- Three tables holding the log K values for the solubilities of the gases O_2(g), H_2(g), and N_2(g). Only the values for O_2(g) are used by the software.

3.2.3 Redox couples

Redox coupling reactions in "oct94" and "feb94" format datasets can be balanced only in terms of "O2(aq)" as the electron acceptor.

3.2.4 Free electron

Databases in the “oct94” and “feb94” format do not include a reaction representing take-up of the free electron. Instead, the software constructs the reaction from the first table mentioned above.

3.2.5 Formulae for aqueous species

Databases prior to the “jul17” format cannot include stoichiometric formulae for the aqueous species.

3.2.6 Fugacity coefficients

Beginning with the “jul17” format, gas species blocks can contain an optional line giving the factors needed to calculate fugacity coefficients by the Spycher-Reed equation, as well as a line holding values used by the Tsonopoulos and Peng-Robinson models.

Report Command

The “report” command returns the results calculated by a GWB program. The command is available in programs **Rxn**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t**. After you have run the program (i.e., after you have selected **Run** → **Go**, or otherwise triggered the calculation), you can use the “report” command to retrieve the program results.

You can use the “report” command from the **Command** pane as a way to explore the calculation results interactively. More commonly, the command is used when writing control scripts (see **Control Scripts**), or when invoking the programs with the plug-in feature (see **Plug-in Feature**) or by remote control (see **Remote Control**), as a method of transmitting calculation results to the script or controlling program.

You might, for example, type

```
report pH
```

on **React**’s **Command** pane. The program would respond with the most recently calculated value of pH. The programs recognize a number of “report” command keywords, like “pH” in the example above; the keywords are listed in the table at the end of this chapter. Typing the command

```
report options
```

returns a list of available keywords.

Depending on the keyword, the “report” command may return a single value, several values, or a vector of values. You control the command’s response using arguments specific to a keyword; arguments are shown in boldface in the table at the end of the chapter. The “concentration” keyword is a good example of using arguments:

concentration <argument(s)> <name(s) | index> . . .

argument:	related arguments:
original	<**fluid** | **system** | **sorbed** | **stagnant**>
current	<**fluid** | **system** | **sorbed**>
aqueous	
surf_species	
elements	<**fluid** | **system** | **sorbed** | **stagnant**>
minerals	

Selecting from the list of types, you can enter a command such as

```
report concentration aqueous
```

which will cause the program to return a vector of the concentrations of aqueous species in the system. The command

```
report aqueous
```

displays the names of those species.

As a second example, typing

```
report concentration original fluid
```

gives a vector of the concentrations of the original basis components in the fluid, the names of which are returned by the command

```
report basis original
```

Notably, the first example above returns concentrations of free species, whereas the second example returns the total or bulk concentrations of the components that make up the solution.

Continuing the first example, you can request the concentration of an individual species by name

```
report concentration aqueous Na+
```

or by vector position

```
report concentration aqueous 12
```

The GWB applications index vectors by offset, so the first entry is identified as "0", the second is "1", and so on.

You can stack arguments on a command line, so typing

```
report concentration aqueous H+ Na+ Cl-
```

prompts the program to return three values, one for each of the species listed. Typing

```
report concentration aqueous 0 1 2
```

also returns three values, for the first three entries in the vector of aqueous species.

In a client application, you may wish to work in terms of vector indices. The number of aqueous species, for example, is returned by the command

```
report naqueous
```

By writing a loop in which a counter *i* varies from zero to the number of aqueous species, less one, you can use the command

```
report aqueous i
report concentration aqueous i
```

to retrieve the names and concentrations of the aqueous species, one at a time.

When using the “report” command, remember to enclose multi-word arguments, such as species names, in quotes, just as you would in any GWB command. For example, the command

```
report mass_reacted "Albite low"
```

gives the expected result.

The “report” command normally returns values in terms of a default unit set for each keyword, as shown in the table at the end of this chapter. To find the default unit for a given “report” keyword, type a command of the form

```
report get_default_units concentration
```

In this case, the application will respond that results for concentration are reported by default in molal units.

You may nonetheless request results in any of the units listed in the **Units Recognized** chapter in this guide. To do so, affix the unit name to the end of a “report” command. For example,

```
report concentration aqueous mmol/kg
```

returns the concentration of each aqueous species, in units of mmol/kg. If the unit conversion fails, the program will respond with “ANULL”, the flag for an undefined value.

You can use the “set_units” keyword to set the application to return results invariably in terms of a specific unit. To override the default units in this way, enter a command such as

```
report set_units "mmol/kg"
```

Having issued this command, unit conversion for commands such as “report temperature” will fail until you have unset the option. To return to default behavior, enter

```
report set_units ?
```

The command

```
report get_units
```

shows the current setting for the overriding unit, if one has been set.

To change the number of significant digits in the numerical results returned by the “report” command, type, for example

```
report set_digits 8
```

By default, the applications return four significant digits.

Finally, in **X1t**, you can specify the node of interest by typing

```
report set_node 5
```

for example. The command above tells the program to return values associated with the node with index 5. Similarly, in **X2t**, you might type

```
report set_node 8 8
```

giving first an x-direction, then y-direction index. Node indices vary from 0 to the total number of nodes, less one. Indexing starts in the bottom left corner of the domain and increases from left to right in the bottom row, then the next highest row, and so on.

(This page left blank.)

<table>
<tr><th>Keyword</th><th>Arguments</th><th>Description</th></tr>
<tr><td>activity</td><td><aqueous | surf_species>
<name(s) | index>. . .</td><td>Species’ activities</td></tr>
<tr><td>alkalinity</td><td></td><td>Alkalinity</td></tr>
<tr><td>aqueous</td><td><index>. . .</td><td>Names of aqueous species</td></tr>
<tr><td>basis</td><td><original | current> <index>. . .</td><td>Names of basis entries</td></tr>
<tr><td>biomass</td><td><reactant(s) | index>. . .</td><td>Biomass concentration</td></tr>
<tr><td>boltzman</td><td><surf_species | index>. . .</td><td>Boltzman factors for surface species</td></tr>
<tr><td>bulk_volume</td><td></td><td>Bulk volume of nodal block</td></tr>
<tr><td>cat_area</td><td><reactant(s) | index>. . .</td><td>Areas of catalyzing surfaces</td></tr>
<tr><td>charge</td><td><type> <name(s) | index>. . .
original | current | aqueous |
surf_species</td><td>Charge on components
or species</td></tr>
<tr><td>chlorinity</td><td></td><td>Chlorinity</td></tr>
<tr><td>coef_dispersion</td><td></td><td>Coefficient of dispersion</td></tr>
<tr><td>concentration</td><td><type> <name(s) | index>. . .
original <fluid | system | sorbed
| stagnant | colloid>
current <fluid | system | sorbed>
aqueous
surf_species
elements <fluid | system | sorbed
| stagnant | colloid>
minerals</td><td>Concentration of components, species, or elements</td></tr>
<tr><td>couples</td><td><index>. . .</td><td>Names of redox couples</td></tr>
<tr><td>discharge</td><td></td><td>Specific discharge</td></tr>
<tr><td>Deltat</td><td></td><td>Length of current time step</td></tr>
<tr><td>EC</td><td></td><td>Electrical conductivity</td></tr>
<tr><td>Eh</td><td><system | couples>
<name(s) | index>. . .</td><td>The system Eh or Nernst Eh values for redox couples</td></tr>
<tr><td>elements</td><td><index>. . .</td><td>Names of elements</td></tr>
</table>

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
	double		✓	✓	✓	✓
mg/kg sol'n as $CaCO_3$	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
	strings		✓	✓	✓	✓
mg/kg	double			✓	✓	✓
	double		✓	✓	✓	✓
cm^3	double		✓	✓	✓	✓
cm^2	double			✓	✓	✓
	double		✓	✓	✓	✓
molal	double		✓	✓	✓	✓
cm^2/s	double				✓	
molal	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
cm^3/cm^2 s	double				✓	✓
s	double			✓	✓	✓
μS/cm or umho/cm	double		✓	✓	✓	✓
volts	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓

Keyword	Arguments	Description
equil_eqn		Equilibrium equation as a text string
equil_favors		Whether reaction products or reactants are favored
equil_temp		Equilibrium temperature
freeflowing		Volume of free-flowing zone in nodal block
FA	<reactant(s) | index>. . .	Kinetic factor for electon acceptance by microbes
FD	<reactant(s) | index>. . .	Kinetic factor for electon donation by microbes
fugacity	<gas(es) | index>. . .	Gas fugacities
gamma	<**aqueous** | **surf_species**> <name(s) | index>. . .	Species' activity coefficients
gas_pressure	<gas(es) | index>. . .	Gas partial pressures
gases	<index>. . .	Names of gases
get_default_units	<keyword>	Default unit for command
get_units		Current unit, if set
hardness		Hardness
hardness_carb		Carbonate
hardness_ncarb		Non-carbonate
hyd_pot		Hydraulic potential
imbalance		Charge imbalance
imbalance_error		Error percentage
inert_volume		Inert volume in system
IS *or* Tionst		System ionic strength
isotopes	<**symbols**>	Names of or symbols for isotope systems
logfO2		Log fugacity of O_2
logk		Log equilibrium constant
logks		List of log K's at principle temperatures

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
	string	✓				
	string	✓				
	string	✓				
cm^3	double		✓	✓	✓	✓
	double			✓	✓	✓
	double			✓	✓	✓
	double		✓	✓	✓	✓
	double		✓	✓	✓	✓
	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
	string	✓	✓	✓	✓	✓
	string	✓	✓	✓	✓	✓
mg/kg sol'n as $CaCO_3$	double		✓	✓	✓	✓
mg/kg sol'n as $CaCO_3$	double		✓	✓	✓	✓
mg/kg sol'n as $CaCO_3$	double		✓	✓	✓	✓
bar	double				✓	✓
eq/kg	double		✓	✓	✓	✓
% error	double		✓	✓	✓	✓
cm^3	double		✓	✓	✓	✓
molal	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
log fugacity	double		✓	✓	✓	✓
	double	✓				
	double	✓				

<table>
<tr><th>Keyword</th><th>Arguments</th><th>Description</th></tr>
<tr><td>**logQoverK *or* SI**</td><td><**minerals** | **reactants**>
<name(s) | index>...</td><td>Saturation index for minerals or reactants</td></tr>
<tr><td>**mass**</td><td><type> <name(s) | index>...
original <**fluid** | **system** | **sorbed** | **stagnant** | **colloid**>
current <**fluid** | **system** | **sorbed**>
aqueous
surf_species
elements <**fluid** | **system** | **sorbed** | **stagnant** | **colloid**>
minerals</td><td>Mass of components, species, or elements</td></tr>
<tr><td>**mass_reacted**</td><td><reactant(s) | index>...</td><td>Mass of a reactant that has reacted</td></tr>
<tr><td>**mass_remaining**</td><td><reactant(s) | index>...</td><td>Mass of a reactant remaining to react</td></tr>
<tr><td>**minerals**</td><td><**current** | **all**> <index>...</td><td>Names of minerals</td></tr>
<tr><td>**mv**</td><td><mineral(s) | index>...</td><td>Mineral molar volume</td></tr>
<tr><td>**mw**</td><td><type> <name(s) | index>...
original | **current** | **aqueous** | **surf_species** | **elements** | **minerals** | **gases**</td><td>Mole weight of components, species, or elements</td></tr>
<tr><td>**naqueous**</td><td></td><td>Number of aqueous species</td></tr>
<tr><td>**nbasis**</td><td></td><td>Number of basis entries</td></tr>
<tr><td>**ncouples**</td><td></td><td>Number of redox couples</td></tr>
<tr><td>**nelements**</td><td></td><td>Number of elements</td></tr>
<tr><td>**ngases**</td><td></td><td>Number of gases</td></tr>
<tr><td>**nisotopes**</td><td></td><td>Number of isotope systems</td></tr>
<tr><td>**nlogks**</td><td></td><td>Number of log K values in list</td></tr>
<tr><td>**nminerals**</td><td><**current** | **all**></td><td>Number of minerals</td></tr>
<tr><td>**Nnode**</td><td></td><td>Number of nodal blocks</td></tr>
<tr><td>**nreactants**</td><td></td><td>Number of reactants, kinetic reactions</td></tr>
<tr><td>**nsorbed**</td><td></td><td>Number of original basis species that sorb</td></tr>
</table>

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
log Q/K	double		✓	✓	✓	✓
moles	double		✓	✓	✓	✓
moles	double			✓	✓	✓
moles	double			✓	✓	✓
	strings		✓	✓	✓	✓
cm^3/mol	double		✓	✓	✓	✓
g/mol	double		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int	✓				
	int		✓	✓	✓	✓
	int				✓	✓
	int			✓	✓	✓
	int		✓	✓	✓	✓

Keyword	Arguments	Description
nsorbing_surfaces		Number of sorbing surface types
nsurf_species		Number of surface species
Nx		Number of nodes along *x*
Ny		Number of nodes along *y*
options		List of keywords for the report command
pe	<**system** | **couples**> <name(s) | index>. . .	The system pe or theoretical pe for redox couples
permeability		Sediment permeability
pH		System pH
polyfit		Coefficients for polynomial fit of log K to temperature
porosity		Porosity
pressure		Pressure
PV		Pore volumes displaced
QoverK	<**minerals** | **reactants**> <name(s) | index>. . .	*Q/K* for a mineral or reactant
rate_con	<reactant(s) | index>. . .	Rate constants for kinetic reactions
reactant_area	<reactant(s) | index>. . .	Surface areas of kinetic/newline minerals
reactants	<index>. . .	Names of reactants and kinetic reactions
reaction		Chemical reaction as a text string
rock_mass		Mass of minerals in system or block
rock_volume		Volume of minerals in system or block

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
	int		✓	✓	✓	✓
	int		✓	✓	✓	✓
	int				✓	✓
	int					✓
	strings	✓	✓	✓	✓	✓
	double		✓	✓	✓	✓
$\mathrm{darcy} \approx \mu\mathrm{m}^2$	double		✓	✓	✓	✓
	double		✓	✓	✓	✓
	doubles	✓				
volume fraction	double		✓	✓	✓	✓
bars	double		✓	✓	✓	✓
	double			✓	✓	✓
	double			✓	✓	✓
(varies)	double			✓	✓	✓
cm^2	double			✓	✓	✓
	strings			✓	✓	✓
	string	✓				
kg	double		✓	✓	✓	✓
cm^3	double		✓	✓	✓	✓

Keyword	Arguments	Description
rxn_rate	<reactant(s) | index>...	Reaction rates
set_digits	<significant digits>	Set number of significant digits in results
set_node	<node index | x y>	Results are associated with the node index or the *x* and *y* node indices
set_units	<unit | ?>	Report results in a unit different than the default
SIS *or* **Sionst**		Stoichiometric ionic strength
soln_density		Solution density
soln_mass		Solution mass
soln_viscosity		Viscosity of fluid
soln_volume		Volume of fluid
sorb_area	<surface_type(s) | index>...	Areas of sorbing surfaces
sorbed	<index>...	Names of original basis species that sorb
sorbing_surfaces		Names of sorbing surface types
stagnant		Volume of stagnant zone in nodal block
success		Returns a value of one if the GWB application has successfully completed a calculation, zero if it has not or if the last calculation failed
surf_charge	<surface_type(s) | index>...	Electrical charge on sorbing surfaces
surf_potential	<surface_type(s) | index>...	Electrical potential of sorbing surfaces
surf_species	<index>...	Names of surface species
TDS		Total dissolved solids
temperature	*or* **T**	Temperature
temps		List of principle temperatures
Tend		Final time of simulation
Time		Current point in time

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
mol/kg s	double			✓	✓	✓
	int	✓	✓	✓	✓	✓
	int				✓	✓
	string	✓	✓	✓	✓	✓
molal	double		✓	✓	✓	✓
g/cm^3	double		✓	✓	✓	✓
kg	double		✓	✓	✓	✓
cp	double		✓	✓	✓	✓
cm^3	double		✓	✓	✓	✓
cm^2	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
	strings		✓	✓	✓	✓
cm^3	double		✓	✓	✓	✓
	int	✓	✓	✓	✓	✓
μC/cm^2	double		✓	✓	✓	✓
mV	double		✓	✓	✓	✓
	strings		✓	✓	✓	✓
mg/kg	double		✓	✓	✓	✓
°C	double		✓	✓	✓	✓
°C	doubles	✓				
s	double			✓	✓	✓
s	double			✓	✓	✓

<table>
<tr><th>Keyword</th><th>Arguments</th><th>Description</th></tr>
<tr><td>total_biomass</td><td></td><td>Biomass in system or block</td></tr>
<tr><td>total_reacted</td><td></td><td>Mass reacted into system or block</td></tr>
<tr><td>TPF</td><td><reactant(s) | index> . . .</td><td>Thermodynamic potential factor</td></tr>
<tr><td>Tstart</td><td></td><td>Beginning time of simulation</td></tr>
<tr><td>velocity</td><td></td><td>Fluid velocity</td></tr>
<tr><td>Watact</td><td></td><td>Activity of water</td></tr>
<tr><td>watertype</td><td></td><td>Ion type of water</td></tr>
<tr><td>Wmass</td><td></td><td>Water mass</td></tr>
<tr><td>xcoef_dispersion</td><td></td><td>x coefficient of dispersion</td></tr>
<tr><td>xdischarge</td><td></td><td>x specific discharge</td></tr>
<tr><td>Xfree</td><td></td><td>Free-flowing fraction</td></tr>
<tr><td>Xi</td><td></td><td>Reaction progress</td></tr>
<tr><td>xpermeability</td><td></td><td>x permeability</td></tr>
<tr><td>xsorbed</td><td><name(s) | index> . . .</td><td>Sorbed fraction of an original basis entry</td></tr>
<tr><td>xvelocity</td><td></td><td>Fluid velocity along x</td></tr>
<tr><td>xycoef_dispersion</td><td></td><td>xy coefficient of dispersion</td></tr>
<tr><td>ycoef_dispersion</td><td></td><td>y coefficient of dispersion</td></tr>
<tr><td>ydischarge</td><td></td><td>y specific discharge</td></tr>
<tr><td>ypermeability</td><td></td><td>y permeability</td></tr>
<tr><td>yvelocity</td><td></td><td>Fluid velocity along y</td></tr>
<tr><td>isotope |
Hydrogen-2 |
Carbon-13 |
Oxygen-18 |
Sulfur-34 |
symbol |
2-H | 13-C |
18-O | 34-S</td><td><fluid | rock | sorbate | system>
solvent
aqueous
minerals
gases
surf_species
reactants
<name(s) | index> . . .</td><td>Isotopic compositions of various aspects of system</td></tr>
</table>

Default units	Return type	Rxn	SpecE8	React	X1t	X2t
mg/kg	double			✓	✓	✓
g	double			✓	✓	✓
	double			✓	✓	✓
s	double			✓	✓	✓
cm/s	double				✓	✓
	double		✓	✓	✓	✓
	string		✓	✓	✓	✓
kg	double		✓	✓	✓	✓
cm^2/s	double					✓
cm^3/cm^2 s	double					✓
	double		✓	✓	✓	✓
	double			✓	✓	✓
$\mathrm{darcy} \approx \mu\mathrm{m}^2$	double					✓
	double		✓	✓	✓	✓
cm/s	double					✓
cm^2/s	double					✓
cm^2/s	double					✓
cm^3/cm^2 s	double					✓
$\mathrm{darcy} \approx \mu\mathrm{m}^2$	double					✓
cm/s	double					✓
δ (‰)	double		✓	✓	✓	✓

Control Scripts

When you read an ordinary script in one of the GWB applications (**Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **Phase2**, **X1t**, or **X2t**), the application steps through the script line-by-line, executing the commands encountered, until it reaches the script's end. A control script differs from an ordinary script in that it can contain control statements, such as assignments, loops, and if-then-else constructs.

This chapter describes how to set up a control script and gives an example of such a script. The **Multiple Analyses** chapter in this guide shows a further example of how control scripts can be applied, in this case to add the results of GWB calculations to a spreadsheet containing the results of a number of chemical analyses.

A control script can occupy an entire script file, or just a portion of one. The control script is preceded by the statement

```
script
```

The control script terminates at the end of the file, or with the statement

```
script end
```

The lines within the control script may be either application commands (e.g., commands recognized by **React**) or control statements.

When writing file names within control scripts, following Tcl syntax, you use double rather than single backslashes (i.e., "\\" instead of "\") as separators. In addition, you enclose filenames containing spaces or special characters with braces ("{. . . }"). For example, the application commands

```
read GWB\react.rea
data "c:\Program Files\GWB\gtdata\thermo.data"
```

would appear as

```
read GWB\\react.rea
data {"c:\\Program Files\\GWB\\gtdata\\thermo.data"}
```

within a control script.

5.1 Control statements

The applications recognize control statements in the form of Tcl commands. Tcl (pronounced “tickle”) is an open-source scripting language, designed to be easy to learn and use. You can find information about the Tcl syntax on various web sites such as www.tcl.tk and mini.net/tcl, as well as a number of widely available textbooks (you can search on “Tcl” at www.amazon.com).

In Tcl, you define a variable with the “set” command, and use a “$” in front of the variable name to reference its value. For example, the commands

```
set pH 4.5
set label "The pH is"
puts "$label $pH"
```

assign a value of 4.5 to a variable named “pH” and a literal string to variable “label”, and then write them to the screen using the “puts” command.

Other useful commands include

`for`	Define a loop
`while`	Define a loop
`eval`	Evaluate a command
`expr`	Evaluate an expression

`if {...}`
`elseif {...}`
`else`
	Set an if-then-else block

`proc`	Define a procedure
`open`	Open a file
`close`	Close a file
`gets`	Get input from a file or prompt
`puts`	Output data to a file or prompt
`info commands`	List Tcl commands

If you write procedures (using the “proc” command), you should be careful not to name them using GWB keywords, or the names of species in the thermo dataset. For example, if you were to name a procedure “Fe++”, you would no longer be able to constrain the concentration of ferrous iron.

Before beginning to write a command script, you will want to consult a more complete Tcl reference to learn a few details of the language syntax.

5.2 Interacting with the application program

You can use the Tcl “eval” command to construct a GWB command and execute it within the GWB application. In a **React** control script, for example, the Tcl command

```
eval {"pH =" $value}
```

causes pH in the **React** run to be set to the contents of variable “value”.

You can interrogate the GWB application program about its calculation results using the “report” command. For example, once you have calculated a geochemical model using **React**, the commmand

```
report pH
```

returns the predicted pH. You can set the value of a variable “new_pH” in a command script to this value with the Tcl command

```
set new_pH [eval report pH]
```

5.3 Example control script

The following control script uses program **React** to search for the rate constant that describes the results of a hydrothermal experiment. In the experiment, 1 kg of water with an initial silica concentration of 1 mg/kg reacts at 100°C with 5000 g of quartz.

After 5 days, the silica concentration is observed to be .55 $m$molal. The script searches for a rate constant in the range 10^{-16} to 10^{-14} mol/cm^2 sec that explains this result. In the script, **React** commands are shown in bold face, for clarity, whereas Tcl commands are shown in normal typeface.

```
time begin = 0 days, end = 5 days
T = 100
SiO2(aq) = 1 mg/kg
react 5000 g Quartz
kinetic Quartz surface = 1000

script start
proc find_ratecon {low high species conc} {
   set gotit 0
   for {set i 0} {$i < 50} {incr i 1} {
      set test [expr {($low + $high) / 2}]
      eval {kinetic Quartz rate_con = $test}
      go
      set back [eval report molality $species]
      if {[expr {abs($back - $conc)}] < 1e-6} {
        set gotit 1
        break
      }
      if {$back < $conc} {
        set low $test
      } else {set high $test}
   }
   if {$gotit} {
```

```
      puts "The optimum rate constant is $test mol/cm2 s"
      puts "The control script converged in $i iterations"
   } else {
      puts "The control script did not converge"
   }
}
# Find the rate constant for Quartz dissolution that gives a
# SiO2(aq) concentration of .00055 molal after 5 days.
find_ratecon 1e-16 1e-14 "SiO2(aq)" .00055
script end
```

5.4 Tcl license agreement

GWB control scripts are evaluated according to the Tcl scripting language, using open source software distributed under the following license agreeement:

Plug-in Feature

The GWB plug-in feature is implemented as a Dynamic-Link Library (DLL). For ease of use, GWB provides wrapper classes for C++, Fortran, Java, Perl, and Python that handle loading the DLL, binding to the needed functions, and conversion to C data types. You might create your own wrapper for the plug-in feature in other languages.

You can plug-in the capabilities of **Rxn**, **SpecE8**, **React**, **Phase2**, **X1t**, and **X2t** to a program that you write using the GWB plug-in feature.

In writing a program of your own, for example, you might need to determine the saturation state of calcite in a fluid of arbitrary composition. Instead of developing code to calculate the distribution of mass and mineral saturation states in a fluid, you could use **SpecE8** from within your program to do the work for you.

Similarly, you could use the plug-in feature to balance reactions with **Rxn**, or figure the results of irreversible reaction paths with **React**.

In each case, you would configure the GWB plug-in with text commands, trigger the calculation with a "go" command, and then retrieve the calculation results to use for your own purposes.

You transfer the results from the GWB plug-in with the "results" function, an interface for the "report" command (documented in the **Report Command** chapter in this **GWB Reference Manual**).

You may also simply read datasets, such as "SpecE8_output.txt", produced by the GWB applications, into your program.

The following sections describe and give examples of how to take advantage of the plug-in feature when developing your software with specific languages and compilers.

6.1 C++

GWB provides a GWBplugin wrapper class contained in the header file "GWBplugin.h", and the GWBplugin.dll export library "GWBplugin.lib" to link to. GWBplugin.h is installed in the "src" subdirectory of the GWB installation directory while the GWBplugin.lib is installed to the main GWB installation directory. In order to locate the GWB DLLs the GWBplugin class uses, you must add the GWB installation directory to the PATH environment variable.

This is the C++ wrapper class provided in GWBplugin.h:

```
// GWBplugin.h

#define ANULL -999999.0        // marker for an undefined value

class GWBplugin {
public:
   GWBplugin();
   ~GWBplugin();
   int initialize(const char* app_name, const char* file_name = NULL,
                 const char* cmds = NULL);
   int exec_cmd(char* uline);
   int results(void* data, const char* value, const char* units = NULL,
                 int ix = 0, int jy = 0);
};
```

6.1.1 Initializing the GWB application

Within your code, first create a GWBplugin object.

```
#include "GWBplugin.h"

GWBplugin myPlugin;
```

Next, use the "initialize" function to start the GWB application of interest by passing the application name, an optional output file name, and any command-line type arguments. The "initialize" function must be called before calling any of the other functions.

```
int initialize (
   const char* app_name,
   const char* file_name = NULL,
   const char* cmds = NULL
);
```

Parameters:

app_name
String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.
file_name (optional)
String containing the name of the file you want the GWB output written to. This can be NULL or an empty string if you do not want the output to be written to a file.
cmds (optional)
String containing command-line options you could normally pass to the application when running it from the command-line. This can be NULL or an empty string.
Command-line options:

`-cd`	Change the working directory to the directory containing the input script specified with the `-i` option.
`-nocd`	Do not change the working directory.
`-i` <input_script>	Read initial input commands from the specified file.
`-gtd` <gtdata_dir>	Set directory to search for thermodynamic datasets.
`-cond` <cond_data>	Set the dataset for calculating electrical conductivity.
`-d` <thermo_data>	Set the thermodynamic dataset.
`-s` <surf_data>	Set a dataset of surface sorption reactions.
`-iso` <isotope_data>	Set a dataset of isotope fractionation factors.

Return value
Non-zero on success and zero on failure.
Examples
Some examples of how to start the GWB plug-in in various ways:

```
// plug-in SpecE8 with no output written and no command-line options
int success = myPlugin.initialize("spece8");

// plug-in React with output written to output.txt and no command-line options
int success = myPlugin.initialize("react", "output.txt");

// plug-in X1t with no output written, no working directory change,
// and input read from pb_contam.x1t
int success = myPlugin.initialize("x1t", NULL, "-nocd
              -i \"c:/program files/gwb/script/pb_contam.x1t\"");
```

6.1.2 Configuring and executing calculations

Use the “exec_cmd” function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a “go” command to trigger the calculations.

```
int exec_cmd (
   char* uline      // command string to be sent to the GWB application
);
```

Return value

Non-zero on success and zero on failure.

Examples

```
myPlugin.exec_cmd("3 mmol H+");
myPlugin.exec_cmd("2 mmol Ca++");
myPlugin.exec_cmd("5 mmolar Cl-");
myPlugin.exec_cmd("go");                    // trigger the calculation
```

6.1.3 Retrieving the results

Transfer calculation results from the GWB application to your program with the "results" function. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual.

To use the "results" function, you provide the address of a data block to fill, along with the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
int results(
   void* data,
   const char* value,
   const char* units = NULL,
   int ix = 0,
   int jy = 0
);
```

Parameters:

data

Address of data block to fill. This can be NULL to determine data block size.

value

String containing the report command keyword and arguments.

units (optional)

String containing the units you want the results returned in. This can be NULL or an empty string if you want the results returned in the default units.

ix (optional)

X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional)

Y node position. This is only used when running X2t, otherwise it is ignored.

Return value

The number of values written (or to be written) to the data block.

Remarks

To determine the size of data block you will need, first call this function with the data parameter as NULL and with the rest of the parameters filled. If you know that the report command you are using only returns a single value, you can simply pass a pointer to the correct data type. See the **Report Command** chapter for details on data types and available keywords.

Examples

```
// get aqueous species names
int ndata = myPlugin.results(NULL, "species");
char** Species = new char*[ndata];
myPlugin.results(Species, "species");

// get aqueous species concentrations in mg/kg
double* Conc = new double[ndata];
myPlugin.results(Conc, "concentration aqueous", "mg/kg");

// get pH at node 3,5
double pH = ANULL;
myPlugin.results(&pH, "pH", NULL, 3, 5);
```

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data will be filled with ANULL (-999999.0).

6.1.4 C++ code examples using the plug-in feature

Normally you would use the GWB plug-in within your program with no output being written to a file. The following is an example of this:

```
#include "GWBplugin.h"

int main(int argc, char* argv[ ])
{
   // create the plug-in object
   GWBplugin* myGWBrun = new GWBplugin();

   // start the GWB program
   if (myGWBrun->initialize("spece8", NULL, "-nocd")) {   // started successfully

      printf("Beginning run.\n");
      // configure SpecE8 and trigger calculation
      const char* cmds[3] = {"pH = 8", "molality Cl- = .05", "go"};
      for(int i = 0; i < 3; i++)
         myGWBrun->exec_cmd(cmds[i]);
      printf("Finished run.\n\n");

      // retrieve results
      double pH;
      myGWBrun->results(&pH, "pH");

      double Cl;
      myGWBrun->results(&Cl, "concentration Cl-");              // in default units
      printf("concentration of Cl- in molal is %10.4g\n", Cl);
      myGWBrun->results(&Cl, "concentration Cl-", "mg/kg");  // in different units
```

```
      printf("concentration of Cl- in mg/kg %10.4g\n", Cl);

      // get size of data
      int nspec = myGWBrun->results(NULL, "species");

      // create data blocks
      const char** Name = new const char*[nspec];
      double* Spec = new double[nspec];

      // send data pointer with keyword and arguments
      myGWBrun->results(Name, "species");
      myGWBrun->results(Spec, "concentration aqueous", "mg/kg");

      printf("\n There are %d aqueous species\n\n", nspec);
      for(int i = 0; i < nspec; i++)
         printf("%-4s = %10.4g mg/kg\n", Name[i], Spec[i]);

      delete[ ] Name;
      delete[ ] Spec;
   }
   else {
      // handle failure to start within your program
   }
   delete myGWBrun;
   return 0;
}
```

To familiarize yourself with the plug-in feature, you might want the GWB program's normal output and results to be written to the console and to text files. The following code shows examples of this:

```
#include "GWBplugin.h"
#include <stdio.h>

int main(int argc, char* argv[ ])
{
   fprintf(stdout, "Starting program SpecE8\n");

   GWBplugin* myGWBrun = new GWBplugin();

    if (myGWBrun->initialize(
      "spece8",
      "test_output.txt",
      "-nocd -i \"C:/Program Files/Gwb/Script/Freshwater.sp8\"
       -s \"C:/Program Files/Gwb/Gtdata/FeOH.sdat\""))
   {                                                    // started successfully

      fprintf(stdout, "writing output to test_output.txt\n");
```

```
myGWBrun->exec_cmd("show surfaces"); // write to output file

fprintf(stdout, "Executing test\n");

FILE *fp;
if ((fp=fopen("test_results.txt", "w")) == NULL) {
   fprintf(stderr, "can't open test_results.txt\n");
}
else {
   fprintf(stdout, "writing results to test_results.txt\n");

   char* cmds[3] = {"pH = 8", "molality Cl- = .05", "go"};
   for(int i = 0; i < 3; i++)
      myGWBrun->exec_cmd(cmds[i]);

   double pH;
   myGWBrun->results(&pH, "pH");
   fprintf(fp, "pH = %4.1f\n", pH);

   double Cl;
   myGWBrun->results(&Cl, "concentration Cl-");
   fprintf(fp, "Cl = %12.5e molal\n", Cl);

   myGWBrun->results(&Cl, "concentration Cl-", "mg/kg");
   if(Cl != ANULL)
      fprintf(fp, "Cl = %12.5e mg/kg\n\n", Cl);
   else
      fprintf(fp, "unit conversion failed - Cl = ANULL\n\n");

   int nspec = myGWBrun->results(NULL, "species");
   const char** Name = new const char*[nspec];
   double* Spec = new double[nspec];

   if (myGWBrun->results(Name, "species")) {
      if (myGWBrun->results(Spec, "concentration aqueous")) {
        for (int i = 0; i < nspec; i++)
           fprintf(fp, "%-32s %12.5e molal\n", Name[i], Spec[i]);
      }
      if (myGWBrun->results(Spec, "concentration aqueous", "mg/kg")) {
           for (int i = 0; i < nspec; i++)
              fprintf(fp, "%-32s %12.5e mg/kg\n", Name[i], Spec[i]);
      }
   }

   if (fp)
      fclose(fp);
```

```
        delete[ ] Name;
        delete[ ] Spec;
      }
   }
   else
      fprintf(stderr, "SpecE8 failed to start\n");

   delete myGWBrun;

   fprintf(stdout, "press return to exit> ");
   getchar();

   return 0;
}
```

6.1.5 C++ compiling and linking

The GWB plug-in has been tested on C++ compilers from Microsoft, Intel, and GCC. The version of the compiler you are using must be the same as the version of GWB installed (32-bit vs. 64-bit).

To compile the GWBplugin Example1 on the command line with the Microsoft or Intel compiler, follow these steps:

```
// open the Microsoft Visual Studio or Intel Command Prompt

// create a working folder and change to that folder
mkdir "%homepath%\GWBplugin"
cd "%homepath%\GWBplugin"

// copy the "src" folder from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\src"

// copy the GWBplugin.lib file from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\gwbplugin.lib"

// add the GWB installation folder to your path
set path=C:\Program Files\GWB;%path%

// compile the example file and tell the compiler to use the GWBplugin library
cl GWBplugin_Cpp_Example1.cpp GWBplugin.lib            // Microsoft
// or
icl GWBplugin_Cpp_Example1.cpp GWBplugin.lib           // Intel

// run the example program
GWBplugin_Cpp_Example1.exe
```

To compile the GWBplugin Example1 on the command line with MinGW, MSYS, and g++, follow these steps:

```
// launch the MinGW shell

// create a working folder and change to that folder
mkdir -p ~/GWBplugin
cd ~/GWBplugin

// copy the "src" folder from GWB installation (default install path shown)
cp /c/program\ files/gwb/src/* .

// copy the GWBplugin.lib file from GWB installation (default install path shown)
cp /c/program\ files/gwb/gwbplugin.lib .

// add the GWB installation folder to your path
PATH=/c/program\ files/gwb:$PATH

// compile the example file and tell the compiler to use the GWBplugin library
g++ GWBplugin_Cpp_Example1.cpp GWBplugin.lib -o
      GWBplugin_Cpp_Example1.exe                    // all on one line

// run the example program
./GWBplugin_Cpp_Example1.exe
```

To compile the GWBplugin Example1 in Microsoft Visual Studio, follow these steps:

```
// open Visual Studio

// create a new project (Ctrl+Shift+N)

// select the "Visual C++" project type and use the "Empty Project" template

// select Project->Add Existing Item... (Shift+Alt+A)
// browse to the "src" subfolder of the GWB installation, select the
// "GWBplugin_Cpp_example1.cpp" file and click "Add"

// open Project->Properties->Configuration Properties

// next to Configuration: select "All configurations"

// next to Platform: select Win32 for 32-bit builds or x64 for 64-bit builds

// under Configuration Properties->

// C/C++->General->Additional Include Directories -
// add the GWB installation "src" folder
// "c:\program files\gwb\src"
```

```
// Linker->Input->Additional Dependencies -
// add the GWBplugin.lib library
// "c:\program files\gwb\gwbplugin.lib"

// Debugging->Environment -
// add the GWB install folder to the path
// path=%path%;c:\program files\gwb

// build using Build->Build Solution

// run using Debug->Start Without Debugging
```

Note: If you need to debug your program you must attach a debugger after the "initialize" call to GWBplugin. A good way to do this is to put in a getchar() call that will pause the program until you can attach the debugger.

6.2 Fortran

GWB provides a Fortran interface, “GWBplugin.f90”, and the GWBplugin.dll export library “GWBplugin.lib” to link to. GWBplugin.f90 is installed in the “src” subdirectory of the GWB installation directory while the GWBplugin.lib is installed in the main GWB installation directory. The GWB plug-in has been tested on Fortran compilers from Intel and GCC. The version of the compiler you are using must be the same as the version of GWB installed (32-bit vs. 64-bit).

This is the Fortran wrapper interface provided in GWBplugin.f90:

```
// GWBplugin.f90

MODULE GWBpluginModule

   INTEGER, PARAMETER :: ANULL = -999999 ! marker for undefined value
   INTEGER, PARAMETER :: GWB_MAX_RESPONSE = 32

   TYPE GWBplugin

    FUNCTION initialize(plugin, app_name, file_name, cmds) RESULT(retval)
     TYPE(GWBplugin), INTENT(out), TARGET :: plugin
     CHARACTER(LEN = *), INTENT(in) :: app_name
     CHARACTER(LEN = *), INTENT(in), OPTIONAL :: file_name, cmds
     INTEGER(C_INT) :: retval

    FUNCTION exec_cmd(plugin, uline) RESULT(retval)
     TYPE(GWBplugin), INTENT(in), TARGET :: plugin
     CHARACTER(LEN = *), INTENT(in) :: uline
     INTEGER(C_INT) :: retval

    FUNCTION results(plugin, f_data, f_value, f_units, ix, jy) RESULT(retval)
     TYPE(GWBplugin), INTENT(in), TARGET :: plugin
     ! f_data :
       CHARACTER(LEN = GWB_MAX_RESPONSE), INTENT(out),
                                OPTIONAL :: f_data(:)
             !or
       REAL(8), INTENT(out), OPTIONAL :: f_data(:)
             !or
       INTEGER, INTENT(out), OPTIONAL :: f_data(:)
     CHARACTER(LEN = *), INTENT(in) :: f_value
     CHARACTER(LEN = *), INTENT(in), OPTIONAL :: f_units
     INTEGER, INTENT(in), OPTIONAL :: ix, jy
     INTEGER(C_INT) :: retval
```

```
SUBROUTINE destroy(plugin)
  TYPE(GWBplugin), INTENT(in), TARGET :: plugin
```

6.2.1 Initializing the GWB application

Within your code, first create a GWBplugin object.

```
INCLUDE "GWBplugin.f90"

USE GWBpluginModule
TYPE(GWBplugin) :: myPlugin
```

Next, use the “initialize” function to start the GWB application of interest by passing the application name, an optional output file name, and any command-line type arguments.

```
FUNCTION initialize(plugin, app_name, file_name, cmds) RESULT(retval)
  TYPE(GWBplugin), INTENT(out), TARGET :: plugin
  CHARACTER(LEN = *), INTENT(in) :: app_name
  CHARACTER(LEN = *), INTENT(in), OPTIONAL :: file_name, cmds
  INTEGER(C_INT) :: retval
```

Parameters:

plugin
An instance of type GWBplugin.

app_name
String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.

file_name (optional)
String containing the name of the file you want the GWB output written to. This can be an empty string if you do not want the output to be written to a file.

cmds (optional)
String containing command-line options you could normally pass to the application when running it from the command-line. This can be an empty string.

Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value

Non-zero on success and zero on failure.

Examples

Some examples of how to start the GWB plug-in in various ways:

```
INTEGER :: success

! plug-in SpecE8 with no output written and no command-line options
success = initialize(myPlugin, "spece8")

! plug-in React with output written to output.txt and no command-line options
success = initialize(myPlugin, "react", "output.txt")

! plug-in X1t with no output written, no working directory change,
! and input read from pb_contam.x1t
success = initialize(myPlugin, "x1t", "",&
             '-nocd -i \"c:/program files/gwb/script/pb_contam.x1t\"')
```

Function "destroy" is used at the end of the program to free up the underlying memory associated with the GWBplugin object.

```
CALL destroy(myPlugin)
```

6.2.2 Configuring and executing calculations

Use the "exec_cmd" function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a "go" command to trigger the calculations.

```
FUNCTION exec_cmd(plugin, uline) RESULT(retval)
   TYPE(GWBplugin), INTENT(in), TARGET :: plugin
   CHARACTER(LEN = *), INTENT(in) :: uline ! command string to be sent
                                          ! to the GWB application
   INTEGER(C_INT) :: retval
```

Return value

Non-zero on success and zero on failure.

Examples

```
err = exec_cmd(myPlugin, "3 mmol H+")
err = exec_cmd(myPlugin, "2 mmol Ca++")
err = exec_cmd(myPlugin, "5 mmolar Cl-")
err = exec_cmd(myPlugin, "go")                ! trigger the calculation
```

6.2.3 Retrieving the results

Transfer calculation results from the GWB application to your program with the “results” function. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual.

To use the “results” function, you provide the address of a data block to fill, along with the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
FUNCTION results(plugin, f_data, f_value, f_units, ix, jy) RESULT(retval)
   TYPE(GWBplugin), INTENT(in), TARGET :: plugin
   ! f_data :
     CHARACTER(LEN = GWB_MAX_RESPONSE), INTENT(out),
                                   OPTIONAL :: f_data(:)
             !or
     REAL(8), INTENT(out), OPTIONAL :: f_data(:)
             !or
     INTEGER, INTENT(out), OPTIONAL :: f_data(:)
   CHARACTER(LEN = *), INTENT(in) :: f_value
   CHARACTER(LEN = *), INTENT(in), OPTIONAL :: f_units
   INTEGER, INTENT(in), OPTIONAL ::  ix, jy
   INTEGER(C_INT) :: retval
```

Parameters:

plugin

An instance of type GWBplugin.

f_data

Address of data block to fill, omit it to find the size of data block needed.

f_value

String containing the report command keyword and arguments.

f_units (optional)

String containing the units you want the results returned in. This can be an empty string if you want the results returned in the default units.

ix (optional)

X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional)

Y node position. This is only used when running X2t, otherwise it is ignored.

Return value

The number of values written (or to be written) to the data block.

Remarks

To determine the size of data block you will need, first call this function with the data parameter omitted and with the rest of the parameters filled. If you know that the report command you are using only returns a single value, you can simply pass an array of size 1 of the correct data type. See the **Report Command** chapter for details on data types and available keywords.

Examples

```
! get aqueous species names
INTEGER :: nspec
nspec = results(myPlugin, "species");
CHARACTER(LEN=GWB_MAX_RESPONSE), ALLOCATABLE :: Species(:)
ALLOCATE(Species(nspec))
results(myPlugin, Species, "species")

! get aqueous species concentrations in mg/kg
REAL(8), ALLOCATABLE :: Conc(:)
ALLOCATE(Conc(nspec))
results(myPlugin, Conc, "concentration aqueous", "mg/kg")

! get pH at node 3,5
REAL(8) :: pH(1)
results(myPlugin, pH, "pH", "", 3, 5)
```

A named constant with the maximum size of strings output by the “results” command is declared in the module:

```
INTEGER, PARAMETER :: GWB_MAX_RESPONSE = 32
```

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data block will be filled with ANULL (-999999).

6.2.4 Fortran code examples using the plug-in feature

The following are Fortran code examples that do the same things as the C++ code examples above. This is a code example using the GWB plug-in within your program with no output being written to a file:

```
INCLUDE "GWBplugin.f90"

PROGRAM FortranPlugin_Example1
   USE GWBpluginModule
   IMPLICIT NONE

   INTEGER :: nspec, err, i
   TYPE(GWBplugin) :: myGWBrun
   CHARACTER (LEN=GWB_MAX_RESPONSE), ALLOCATABLE :: Name(:)
   REAL(8), ALLOCATABLE :: Spec(:)
   REAL(8) :: pH(1), Cl(1)
   CHARACTER(LEN=255), dimension(3), PARAMETER :: &
      cmds = [character(len=255) :: &
      "pH = 8", "molality Cl- = .05", "go"]

   !  create and start the GWB program
```

```
IF (initialize(myGwbrun, "spece8", "", "-nocd") /= 0) THEN

   ! started successfully

   WRITE(*,*) "Beginning run."

   ! configure SpecE8 and trigger calculation
   DO i = 1, 3
      err = exec_cmd(myGwbrun, cmds(i))
   END DO

   WRITE(*,*) "Finished run."
   WRITE(*,*)

   ! retrieve results
   err = results(myGwbrun, pH, "pH")
   err = results(myGwbrun, Cl, "concentration Cl-")
   WRITE(*,'("concentration of Cl- in molal is ", G10.3)') Cl
   err = results(myGwbrun, Cl, "concentration Cl-", "mg/kg")
   WRITE(*,'("concentration of Cl- in mg/kg is ", G10.3)') Cl
   WRITE(*,*)

   ! get size of data
   nspec = results(myGwbrun, "species");
   WRITE(*,'(There are", I2, " aqueous species")') nspec
   WRITE(*,*)

   ! create data blocks
   ALLOCATE(Name(nspec))
   ALLOCATE(Spec(nspec))

   ! send data pointer with keyword and arguments
   err = results(myGwbrun, Name, "species")
   err = results(myGwbrun, Spec, "concentration aqueous", "mg/kg")

   DO i = 1, nspec
      WRITE(*,'(A4, " = ", G10.3, " mg/kg")') Name(i), Spec(i)
   END DO
   WRITE(*,*)

   ! use the data in your program

   DEALLOCATE(Name)
   DEALLOCATE(Spec)
ELSE
   ! handle failure to start within your program
END IF
```

```
   CALL destroy(myGwbrun)

END PROGRAM FortranPlugin_Example1
```

Code example with the GWB application's normal output and results written to the console or to text files:

```
INCLUDE "GWBplugin.f90"

PROGRAM FortranPlugin_Example2
   USE GWBpluginModule
   IMPLICIT NONE

   INTEGER :: nspec, err, i
   TYPE (GWBplugin) :: myGWBrun
   CHARACTER (LEN=GWB_MAX_RESPONSE), ALLOCATABLE :: Name(:)
   REAL(8), ALLOCATABLE :: Spec(:)
   REAL(8) :: pH(1), Cl(1)
   CHARACTER(LEN=255), dimension(3), PARAMETER :: &
      cmds = [character(len=255) :: &
      "pH = 8", "molality Cl- = .05", "go"]
   CHARACTER :: a

   WRITE(*,*) "Starting program SpecE8"

   IF (initialize(myGwbrun, "spece8" , "test_output.txt",&
         '-nocd -i "C:/Program Files/Gwb/Script/Freshwater.sp8"&
         &-s "C:/Program Files/Gwb/Gtdata/FeOH.sdat"') /= 0) THEN

      err = exec_cmd(myGwbrun, "show surfaces")

      WRITE(*,*) "Executing Test"

      OPEN(10, file = 'test_results.txt')
      WRITE(*,*) "Writing results to test_results.txt"

      DO i = 1, 3
         err = exec_cmd(myGwbrun, cmds(i))
      END DO

      err = results(myGwbrun, pH, "pH")
      WRITE(10, '("pH = ", F4.1)') pH

      err = results(myGwbrun, Cl, "concentration Cl-")
      WRITE(10, '("Cl = ", ES12.5, " molal")') Cl

      err = results(myGwbrun, Cl, "concentration Cl-", "mg/kg")
```

```
    IF (err /= ANULL) THEN
       WRITE(10, '("Cl = ", ES12.5, " mg/kg")') Cl
    ELSE
       WRITE(10,*) "unit conversion failed - CL = ANULL"
    END IF

    WRITE(10,*)

    nspec = results(myGwbrun, "species");

    ALLOCATE(Name(nspec))
    ALLOCATE(Spec(nspec))

    IF (results(myGwbrun, Name, "species") /= 0) THEN
       IF (results(myGwbrun, Spec, "concentration aqueous") /= 0) THEN
          DO i = 1, nspec
             WRITE(10, '(A32, ES12.5, " molal")') Name(i), Spec(i)
          END DO
       END IF

       WRITE(10,*)
       WRITE(10,*)

     IF (results(myGwbrun, Spec, "concentration aqueous", "mg/kg") /= 0) THEN
          DO i = 1, nspec
             WRITE(10, '(A32, ES12.5, " mg/kg")') Name(i), Spec(i)
          END DO
       END IF
    END IF

    CLOSE(10)

    DEALLOCATE(Name)
    DEALLOCATE(Spec)

  END IF

  CALL destroy(myGwbrun)

  WRITE(*,*) "enter any letter to exit> "
  READ(*,*) a

END PROGRAM FortranPlugin_Example2
```

6.2.5 Fortran compiling

The GWB plug-in has been tested on Fortran compilers from Intel and GCC. The version of the compiler you are using must be the same as the version of GWB installed (32-bit vs. 64-bit).

To compile the GWBplugin Example1 on the command line with Intel's Fortran compiler, follow these steps:

```
! open the Intel Command Prompt

! create a working folder and change to that folder
mkdir "%homepath%\GWBplugin"
cd "%homepath%\GWBplugin"

! copy the "src" folder from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\src"

! copy the "GWBplugin.lib" file from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\gwbplugin.lib"

! add the GWB installation folder to your path
set path=C:\Program Files\GWB;%path%

! compile the example file and tell the compiler to use the GWBplugin library
ifort GWBplugin_Fortran_Example1.f90 GWBplugin.lib

! run the example
GWBplugin_Fortran_Example1.exe
```

To compile the GWBplugin Example1 on the command line with MinGW, MSYS, and gfortran, follow these steps:

```
! launch the MinGW Shell

! create a working folder and change to that folder
mkdir -p ~/GWBplugin
cd ~/GWBplugin

! copy the "src" folder from GWB installation (default install path shown)
cp /c/program\ files/gwb/src/* .

! copy the "GWBplugin.lib" file from GWB installation (default install path shown)
cp /c/program\ files/gwb/gwbplugin.lib .

! add the GWB installation folder to your path
PATH=/c/program\ files/gwb:$PATH
```

```
! compile the example file and tell the compiler to use the GWBplugin library
gfortran GWBplugin_Fortran_Example1.f90 GWBplugin.lib -o
         GWBplugin_Fortran_Example1.exe                  ! all on one line

! run the example
./GWBplugin_Fortran_Example1.exe
```

To compile the GWBplugin Example1 in Microsoft Visual Studio with Intel's Fortran compiler, follow these steps:

```
! open Visual Studio

! create a new project (Ctrl+Shift+N)

! select the "Intel(R) Visual Fortran" project type,
! select "Console Application", and use the "Empty Project" template

! select Project->Add Existing Item... (Shift+Alt+A)
! browse to the "src" subfolder of the GWB installation,
! select the "GWBplugin_Fortran_example1.f90" file and click "Add"

! open Project->Properties->Configuration Properties

! next to Configuration: select "All configurations"

! next to Platform: select Win32 for 32-bit builds or x64 for 64-bit builds

! under Configuration Properties->

! Fortran->General->Additional Include Directories -
! add the GWB installation "src" folder
! "c:\program files\gwb\src"

! Linker->Input->Additional Dependencies -
! add the GWBplugin.lib library
! "c:\program files\gwb\gwbplugin.lib"

! Debugging->Environment -
! add the GWB install folder to the path
! path=%path%;c:\program files\gwb

! build using Build->Build Solution

! run using Debug->Start Without Debugging
```

Note: If you need to debug your program you must attach a debugger after the "initialize" call to GWBplugin, otherwise your program will encounter a run-time error.

6.3 Java

GWB provides a GWBplugin wrapper class contained in the file "GWBplugin.java" installed in the "src" subdirectory of the GWB installation directory. In order to locate the GWB DLLs the GWBplugin class uses, you must add the GWB installation directory to the PATH environment variable.

To compile your program you will need to have a Java Development Kit (JDK) installed. The version of the Java virtual machine must match the version of GWB installed (32-bit vs. 64-bit). For loading the DLL and conversion to C data types, the GWBplugin class depends on the Java Native Access library (JNA). For ease of use the "GWBplugin.java" wrapper and the JNA library have been combined into the "GWBplugin.jar" file installed in the "src" directory of the GWB installation directory. This jar file must be added to the CLASSPATH variable when compiling.

```
// GWBplugin.java

package GWBplugin;

import com.sun.jna.*;
import com.sun.jna.ptr.PointerByReference;
```

This is the Java wrapper class provided by GWBplugin.java in the "src" subdirectory of the GWB installation directory:

```
public class GWBplugin {

   static public double ANULL = -999999;

   public GWBplugin();
   public int initialize(String app_name, String file_name = null,
                         String cmds = null);
   public int exec_cmd(String uline);
   public int results(Object data, String value, String units = null,
                         int ix = 0, int jy = 0);
   public void destroy();
}
```

6.3.1 Initializing the GWB application

Within your code, first create a GWBplugin object.

```
import GWBplugin.GWBplugin;

GWBplugin myPlugin = new GWBplugin();
```

Next, use the “initialize” function to start the GWB application of interest by passing the application name, an optional output file name, and any command-line type arguments. The “initialize” function must be called before calling any of the other functions.

```
public int initialize (
   String app_name,
   String file_name = null,
   String cmds = null
);
```

Parameters:
app_name
String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.
file_name (optional)
String containing the name of the file you want the GWB output written to. This can be null or an empty string if you do not want the output to be written to a file.
cmds (optional)
String containing command-line options you could normally pass to the application when running it from the command-line. This can be null or an empty string.
Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value
Non-zero on success and zero on failure.
Examples
Some examples of how to start the GWB plug-in in various ways:

```
// plug-in SpecE8 with no output written and no command-line options
int success = myPlugin.initialize("spece8");

// plug-in React with output written to output.txt and no command-line options
int success = myPlugin.initialize("react", "output.txt");

// plug-in X1t with no output written, no working directory change,
// and input read from pb_contam.x1t
```

```
int success = myPlugin.initialize("x1t", null, "-nocd
            -i \"c:/program files/gwb/script/pb_contam.x1t\"");
```

Function "destroy" can be used at the end of the program to free up the underlying memory associated with the GWBplugin object.

```
myPlugin.destroy();
```

6.3.2 Configuring and executing calculations

Use the “exec_cmd” function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a “go” command to trigger the calculations.

```
public int exec_cmd (
   String uline     // command string to be sent to the GWB application
);
```

Return value

Non-zero on success and zero on failure.

Examples

```
myPlugin.exec_cmd("3 mmol H+");
myPlugin.exec_cmd("2 mmol Ca++");
myPlugin.exec_cmd("5 mmolar Cl-");
myPlugin.exec_cmd("go");                    // trigger the calculation
```

6.3.3 Retrieving the results

Transfer calculation results from the GWB application to your program with the “results” function. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual.

To use the “results” function, you provide an array of the proper data type, along with the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
public int results(
   Object data,
   String value,
   String units = null,
   int ix = 0,
   int jy = 0
);
```

Parameters:

data
Array of data to fill. This can be null or of type int[ ], double[ ], or String[ ].
value
String containing the report command keyword and arguments.
units (optional)
String containing the units you want the results returned in. This can be null or an empty string if you want the results returned in the default units.
ix (optional)
X node position. This is only used when running X1t and X2t, otherwise it is ignored.
jy (optional)
Y node position. This is only used when running X2t, otherwise it is ignored.
Return value
The number of values written (or to be written) to the array.
Remarks
To determine the size of array you will need, first call this function with the data parameter as null and with the rest of the parameters filled.

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data will be filled with GWBplugin.ANULL (-999999).
Examples

```
// get aqueous species names
int ndata = myPlugin.results(null, "species");
String Species[ ] = new String[ndata];
myPlugin.results(Species, "species");

// get aqueous species concentrations in mg/kg
double Conc[ ] = new double[ndata];
myPlugin.results(Conc, "concentration aqueous", "mg/kg");

// get pH at node 3,5
double pH[ ] = new double[1];
myPlugin.results(pH, "pH", null, 3, 5);
```

6.3.4 Java code examples using the plug-in feature

Normally you would use the GWB plug-in within your program with no output being written to a file. The following is an example of this:

```
import GWBplugin.GWBplugin;

// run with "java -Xss10m Example1"
// to avoid possible stack_overflow_exception

class Example1 {

   public static void main(String[ ] args) {
```

```
// create the plug-in object
GWBplugin myGWBrun = new GWBplugin();

// start the GWB program
if(myGWBrun.initialize("spece8", "", "-nocd") != 0) {
   // started successfully
   System.out.println("Beginning run.");

   // configure SpecE8 and trigger calculation
   String[ ] cmds = {"pH = 8", "molality Cl- = .05", "go"};
   for(int i=0; i<3; i++)
      myGWBrun.exec_cmd(cmds[i]);

   System.out.println("Finished run.");

   // retrieve results
   double pH[ ] = new double[1];
   myGWBrun.results(pH, "pH");

   double Cl[ ] = new double[1];
   // in default units
   myGWBrun.results(Cl, "concentration Cl-");
   System.out.println(String.format("concentration of Cl- in molal
                    is %10.4g" ,Cl[0]));
   // in different units
   myGWBrun.results(Cl, "concentration Cl-", "mg/kg");
   System.out.println(String.format("concentration of Cl- in mg/kg
                    is %10.4g", Cl[0]));

   // get size of data
   int nspec = myGWBrun.results(null, "species");

   // create data blocks
   String Name[ ] = new String[nspec];
   double Spec[ ] = new double[nspec];

   // send data arrays with keyword and arguments
   myGWBrun.results(Name, "species");
   myGWBrun.results(Spec, "concentration aqueous", "mg/kg");

   System.out.println(String.format("There are %d aqueous species.",
                    nspec));

   for(int i=0; i<nspec; i++)
      System.out.println(String.format("%-4s = %10.4g mg/kg",
                    Name[i], Spec[i]));
```

```
            myGWBrun.destroy();
         }
      }
   }
```

To familiarize yourself with the plug-in feature, you might want the GWB program's normal output and results to be written to the console and to text files. The following code shows examples of this:

```
   import GWBplugin.GWBplugin;
   import java.io.*;

   // run with "java -Xss10m Example2"
   // to avoid possible stack_overflow_exception

   class Example2 {
      public static void main(String[ ] args) {

         try {
            System.out.println("Starting program SpecE8");

            GWBplugin myGWBrun = new GWBplugin();

            if(myGWBrun.initialize("spece8",
                        "test_output.txt",
                        "-nocd -i \"c:/program files/gwb/script/freshwater.sp8\"
                         -s \"c:/program files/gwb/gtdata/feoh.sdat\"") != 0) {
               // started successfully
               System.out.println("writing output to test_output.txt.");

               myGWBrun.exec_cmd("show surfaces"); // write to output file

               System.out.println("Executing test");

               FileOutputStream fos;
               PrintStream fp;

               fos = new FileOutputStream("test_results.txt");
               fp = new PrintStream(fos);
               System.out.println("writing results to test_results.txt");

               String[ ] cmds = {"pH = 8", "molality Cl- = .05", "go"};
               for(int i=0; i<3; i++)
                  myGWBrun.exec_cmd(cmds[i]);

               double pH[ ] = new double[1];
               myGWBrun.results(pH, "pH");
               fp.println(String.format("pH = %4.1f ", pH[0]));
```

```
        double Cl[ ] = new double[1];
        myGWBrun.results(Cl, "concentration Cl-");
        fp.println(String.format("Cl = %12.5e molal", Cl[0]));
        myGWBrun.results(Cl, "concentration Cl-", "mg/kg");
        if(Cl[0] != GWBplugin.ANULL)
           fp.println(String.format("Cl = %12.5e mg/kg", Cl[0]));
        else
           fp.println("unit conversion failed - Cl = ANULL");
        fp.println("");

        // get size of data
        int nspec = myGWBrun.results(null, "species");

        // create data blocks
        String Name[ ] = new String[nspec];
        double Spec[ ] = new double[nspec];

        // send data arrays with keyword and arguments
        if(myGWBrun.results(Name, "species") != 0) {
           if(myGWBrun.results(Spec, "concentration aqueous") != 0){
              for(int i=0; i<nspec; i++)
                 fp.println(String.format("%-32s %12.5e molal",
                         Name[i], Spec[i]));
           }
           fp.println("");
           fp.println("");
           if(myGWBrun.results(Spec, "concentration aqueous", "mg/kg") != 0)
           {
              for(int i=0; i<nspec; i++)
                 fp.println(String.format("%-32s %12.5e mg/kg",
                         Name[i], Spec[i]));
           }
        }
        fp.close();

        myGWBrun.destroy();

        BufferedReader br = new BufferedReader(new
                                 InputStreamReader(System.in));
        String input = null;
        System.out.println("press return to exit> ");
        input = br.readLine();
     }
  }
  catch (Exception e)
  {
     e.printStackTrace();
```

```
        }
    }
}
```

6.3.5 Java command line

This example of how to run the GWBplugin Example1 on the command line with Java assumes that you have a Java development kit installed.

To run Example1 on the command line with Java, follow these steps:

```
// open the command prompt
cmd.exe

// create a working folder and change to that folder
mkdir "%homepath%\GWBplugin"
cd "%homepath%\GWBplugin"

// copy the "src" folder from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\src"

// add the GWB installation folder to your path
set path=C:\Program Files\GWB;%path%

// add the JDK bin folder to your path if it is not already there
set path=C:\Program Files\Java\jdk1.7.0_05\bin;%path%

// create a build folder
mkdir class

// add the build folder and the GWBplugin JAR file to your classpath
set classpath=class;C:\Program Files\GWB\src\GWBplugin.jar;%classpath%

// compile the example file
javac GWBplugin_Java_Example1.java -d class

// run the example with Java
// (-Xss 10m increases the stack size, the default stack is usually too small)
java -Xss10m Example1
```

6.4 Perl

GWB provides a GWBplugin wrapper class contained in the Perl module file “GWBplugin.pm” which handles dealing with the C data type conversion and calling the DLL. In order to locate the GWB DLLs the GWBplugin class uses, you must add the GWB installation directory to the PATH environment variable.

Since Perl is a dynamically typed language, there are some minor differences with its "results" functions compared to statically typed languages.

To use the GWBplugin class from the GWBplugin module, you must first add the "src" folder in the GWB installation to Perl’s module search path and then tell it to use the GWBplugin module.

```
#! /usr/bin/perl -w

## add explicit location of GWBplugin.pm to lib
use lib '/program files/gwb/src';

## or by relative path
use lib '.';

## use GWBplugin module
use GWBplugin;
```

The GWBplugin module depends on the Perl Win32::API module. You can install the Win32::API module with the Perl Package Manager with the following command:

```
ppm install Win32-API
```

This is the Perl wrapper class provided in GWBplugin.pm in the "src" directory of the GWB installation folder:

```
# GWBplugin.pm

package GWBplugin;

our $ANULL = -999999;

sub initialize      # (app_name, file_name = 0, cmds = 0)
sub exec_cmd        # (uline)
sub results         # (value, units = 0, ix = 0, jy = 0)
sub destroy         #
```

6.4.1 Initializing the GWB application

Within your code, first create a GWBplugin object.

```
my $myPlugin = new GWBplugin();
```

Next, use the “initialize” function to start the GWB application of interest by passing the application name, an optional output file name, and any command-line type arguments. The “initialize” function must be called before calling any of the other functions.

```
sub initialize # (
    app_name,
    file_name = 0,
    cmds = 0)
```

Parameters:

app_name

String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.

file_name (optional)

String containing the name of the file you want the GWB output written to. This can be a zero or an empty string if you do not want the output to be written to a file.

cmds (optional)

String containing command-line options you could normally pass to the application when running it from the command-line. This can be a zero or an empty string for defaults.

Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value

Non-zero on success and zero on failure.

Examples

```
# plug-in SpecE8 with no output written and no command-line options
my $success = $myPlugin->initialize("spece8");

# plug-in React with output written to output.txt and no command-line options
my $success = $myPlugin->initialize("react", "output.txt");

# plug-in X1t with no output written, no working directory change,
```

```
# and input read from pb_contam.x1t
my $success = $myPlugin->initialize("x1t", "", "-nocd
            -i \"c:/program files/gwb/script/pb_contam.x1t\"");
```

Function "destroy" can be used at the end of the program to free up the underlying memory associated with the GWBplugin object.

```
$myPlugin->destroy();
```

6.4.2 Configuring and executing calculations

Use the “exec_cmd” function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a “go” command to trigger the calculations.

```
sub exec_cmd # (
   uline     # command string to be sent to the GWB application
)
```

Return value

Non-zero on success and zero on failure.

Examples

```
$myPlugin->exec_cmd("3 mmol H+");
$myPlugin->exec_cmd("2 mmol Ca++");
$myPlugin->exec_cmd("5 mmolar Cl-");
$myPlugin->exec_cmd("go");                    # trigger the calculation
```

6.4.3 Retrieving the results

Transfer calculation results from the GWB application to your program with the “results” functions. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual. Use the "results" functions by providing the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
# results function
sub results # (value, units = 0, ix = 0, jy = 0)
```

Parameters:

value

String containing the report command keyword and arguments.

units (optional)

String containing the units you want the results returned in. This can be a zero or an empty string if you want the results returned in the default units.

ix (optional)

X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional)

Y node position. This is only used when running X2t, otherwise it is ignored.

Return value

Array containing the requested results.

Remarks

The data is returned as an array, even when requesting a single value.

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data will be filled with ANULL (-999999).

Examples

```
# get aqueous species names
my @species = $myPlugin->results("species");

# get aqueous species concentrations in mg/kg
my @conc = $myPlugin->results("concentration aqueous", "mg/kg");

# get pH at node 3,5
my ($pH) = $myPlugin->results("pH", "", 3,5);
```

6.4.4 Perl code examples using the plug-in feature

Normally you would use the GWB plug-in within your program with no output being written to a file. The following is an example of this:

```
#! /usr/bin/perl -w

## add explicit location of GWBplugin.pm to lib
#   use lib '/program files/gwb/src';

## or by relative path
use lib '.';

## use GWBplugin module
use GWBplugin;

# create the plug-in object
my $myGWBrun = new GWBplugin();

# start the GWB program
if ($myGWBrun->initialize("spece8", "", "-nocd")) {

   print "Beginning run.\n";

   my @cmds = ("pH = 8", "molality Cl- = .05", "go");
   foreach my $cmd (@cmds) {
```

```
        $myGWBrun->exec_cmd($cmd);
    }
    print "Finished run.\n\n";

    # retrieve results
    my ($pH) = $myGWBrun->results("pH");

    my ($Cl) = $myGWBrun->results("concentration Cl-");
    printf("concentration of Cl- in molal is %10.4g\n", $Cl);
    ($Cl) = $myGWBrun->results("concentration Cl-", "mg/kg");
    printf("concentration of Cl- in mg/kg is %10.4g\n", $Cl);

    my @species = $myGWBrun->results("species");
    my @conc = $myGWBrun->results("concentration aqueous", "mg/kg");
    my $nspec = @species;

    print "\nThere are " . $nspec . " aqueous species.\n\n";

    for(my $i=0; $i<$nspec; $i++) {
        printf("%-4s = %10.4g mg/kg\n", $species[$i], $conc[$i]);
    }

    $myGWBrun->destroy();
}
```

To familiarize yourself with the plug-in feature, you might want the GWB program's normal output and results to be written to the console and to text files. The following code shows examples of this:

```
#! /usr/bin/perl -w

## add explicit location of GWBplugin.pm to lib
#  use lib '/program files/gwb/src';

## or by relative path
use lib '.';

## use GWBplugin module
use GWBplugin;

print "Starting program SpecE8\n";

my $myGWBrun = new GWBplugin();

if ($myGWBrun->initialize("spece8",
            "test_output.txt",
            "-nocd \
```

```
            -i \"c:/program files/gwb/script/freshwater.sp8\" \
            -s \"c:/program files/gwb/gtdata/feoh.sdat\"")) {

    print "writing output to test_output.txt\n";
    $myGWBrun->exec_cmd("show surfaces");
    print "Executing test\n";
    open FP, ">test_results.txt" or die $!;
    print "writing results to test_results.txt\n";
    my @cmds = ("pH = 8", "molality Cl- = .05", "go");
    foreach my $cmd (@cmds){
       $myGWBrun->exec_cmd($cmd);
    }

    my ($pH) = $myGWBrun->results("pH");
    printf FP ("pH = %4.1f\n", $pH);

    my ($Cl) = $myGWBrun->results("concentration Cl-");
    printf FP ("Cl = %12.5e molal\n", $Cl);

    ($Cl) = $myGWBrun->results("concentration Cl-", "mg/kg");
    if($Cl ne $ANULL) {
       printf FP ("Cl = %12.5e mg/kg\n\n", $Cl);
    }
    else {
       print FP "unit conversion failed - Cl = ANULL\n\n";
    }

    my @Name = $myGWBrun->results("species");
    my @Spec = $myGWBrun->results("concentration aqueous");
    my $nspec = @Name;
    for(my $i=0; $i<$nspec; $i++) {
       printf FP ("%-32s %12.5e molal\n", $Name[$i], $Spec[$i]);
    }
    @Spec = $myGWBrun->results("concentration aqueous", "mg/kg");
    print FP "\n\n";
    for (my $i=0; $i<$nspec; $i++) {
       printf FP ("%-32s %12.5e mg/kg\n", $Name[$i], $Spec[$i]);
    }

    $myGWBrun->destroy();
    print "press return to exit> ";
    <>;
    close(FP);
}
```

6.4.5 Perl command line

This example of how to run the GWBplugin Example1 on the command line with Perl assumes that you have (64-bit) ActivePerl for Windows installed. This example should work with other versions of Perl, but instructions on how to obtain the Win32::API module may be different.

To run Example1 on the command line with Perl, follow these steps:

```
# open the command prompt
cmd.exe

# create a working folder and change to that folder
mkdir "%homepath%\GWBplugin"
cd "%homepath%\GWBplugin"

# copy the "src" folder from GWB installation (default install path shown)
copy /Y "C:\Program Files\GWB\src"

# add the GWB installation folder to your path
set path=C:\Program Files\GWB;%path%

# if you haven't already installed the Win32::API module, do so now
ppm install Win32-API

# run the example with Perl
perl GWBplugin_Perl_Example1.pl
```

6.5 Python

GWB provides a GWBplugin wrapper class contained in the Python script file "GWBplugin.py" which handles dealing with the C data type conversion and calling the DLL. In order to locate the GWB DLLs the GWBplugin class uses, you must add the GWB installation directory to the PATH environment variable.

Since Python is a dynamically typed language, there are some minor differences with its "results" functions compared to statically typed languages.

To include GWBplugin.py in your Python script, you first need to append the "src" folder of the GWB installation to sys.path in Python, then import the class.

```
import os,sys

## append full path to GWBplugin.py ...
sys.path.append("c:/program files/gwb/src")

## or relative path ...
sys.path.append(os.path.abspath('.'))

# import GWBplugin class
from GWBplugin import *
```

This is the Python wrapper class provided in GWBplugin.py in the "src" directory of the GWB installation folder:

```
# GWBplugin.py

ANULL = -999999

class GWBplugin:
   Name = "GWBplugin"
   def _ _init_ _(self):
   def initialize (self, app_name, file_name = None, cmds = None):
   def exec_cmd (self, uline):
   def results (self, value, units = None, ix = 0, jy = 0):
   def destroy (self):
```

6.5.1 Initializing the GWB application

Within your code, first create a GWBplugin object.

```
myPlugin = GWBplugin()
```

Next, use the "initialize" function to start the GWB application of interest by passing the application name, an optional output file name, and any command-line type arguments. The “initialize” function must be called before calling any of the other functions.

```
def initialize (
    self,
    app_name,
    file_name = None,
    cmds = None):
```

Parameters:
app_name
String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.
file_name (optional)
String containing the name of the file you want the GWB output written to. This can be None or an empty string if you do not want the output to be written to a file.
cmds (optional)
String containing command-line options you could normally pass to the application when running it from the command-line. This can be None or an empty string for defaults.

Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value
Non-zero on success and zero on failure.
Examples

```
# plug-in SpecE8 with no output written and no command-line options
success = myPlugin.initialize("spece8")

# plug-in React with output written to output.txt and no command-line options
success = myPlugin.initialize("react", "output.txt")

# plug-in X1t with no output written, no working directory change,
# and input read from pb_contam.x1t
success = myPlugin.initialize("x1t", "", "-nocd
            -i \"c:/program files/gwb/script/pb_contam.x1t\"")
```

Function “destroy" can be used at the end of the program to free up the underlying memory associated with the GWBplugin object.

```
myPlugin.destroy()
```

6.5.2 Configuring and executing calculations

Use the “exec_cmd” function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a “go” command to trigger the calculations.

```
def exec_cmd (
   self,
   uline     # command string to be sent to the GWB application
):
```

Return value

Non-zero on success and zero on failure.

Examples

```
myPlugin.exec_cmd("3 mmol H+")
myPlugin.exec_cmd("2 mmol Ca++")
myPlugin.exec_cmd("5 mmolar Cl-")
myPlugin.exec_cmd("go")                    # trigger the calculation
```

6.5.3 Retrieving the results

Transfer calculation results from the GWB application to your program with the “results” functions. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual. Use the “results” functions by providing the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
# results function
def results (self, value, units = None, ix = 0, jy = 0):
```

Parameters:

value

String containing the report command keyword and arguments.

units (optional)

String containing the units you want the results returned in. This can be None or an empty string if you want the results returned in the default units.

ix (optional)

X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional)

Y node position. This is only used when running X2t, otherwise it is ignored.

Return value

Array containing the requested results.

Remarks

The data is returned as an array, even when requesting a single value.

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data will be filled with ANULL (-999999).

Examples

```
# get aqueous species names
Species = myPlugin.results("species")

# get aqueous species concentrations in mg/kg
Conc = myPlugin.results("concentration aqueous", "mg/kg")

# get pH at node 3,5
pH = myPlugin.results("pH", "", 3,5)[0]
```

6.5.4 Python code examples using the plug-in feature

Normally you would use the GWB plug-in within your program with no output being written to a file. The following is an example of this:

```
import os,sys

## append full path to GWBplugin.py ...
#  sys.path.append("c:/program files/gwb/src")

## or relative path ...
sys.path.append(os.path.abspath('.'))

# import GWBplugin class
from GWBplugin import *

# create the plug-in object
myGWBrun = GWBplugin()

# start the GWB program
if myGWBrun.initialize("spece8", "", "-nocd"):

   print "Beginning run."

   cmds = ["pH = 8", "molality Cl- = .05", "go"]
   for cmd in cmds:
      myGWBrun.exec_cmd(cmd)

   print "Finished run.\n"

```

```
    #retrieve results
    pH = myGWBrun.results("pH")[0]

    Cl = myGWBrun.results("concentration Cl-")[0]
    print "concentration of Cl- in molal is %10.4g" % Cl
    Cl = myGWBrun.results("concentration Cl-", "mg/kg")[0]
    print "concentration of Cl- in mg/kg is %10.4g" % Cl

    species = myGWBrun.results("species")
    conc = myGWBrun.results("concentration aqueous", "mg/kg")

    print "\nThere are" , len(species) , "aqueous species.\n"
    for i in range(len(species)):
        print "%-4s = %10.4g mg/kg" % (species[i], conc[i])

    myGWBrun.destroy()
```

To familiarize yourself with the plug-in feature, you might want the GWB program's normal output and results to be written to the console and to text files. The following code shows examples of this:

```
import os,sys

## append full path to GWBplugin.py ...
#   sys.path.append("c:/program files/gwb/src")

## or relative path ...
sys.path.append(os.path.abspath('.'))

# import GWBplugin class
from GWBplugin import *

print "Starting program SpecE8"

myGWBrun = GWBplugin()

if myGWBrun.initialize("spece8",
              "test_output.txt",
              "-nocd \
              -i \"c:/program files/gwb/script/freshwater.sp8\" \
              -s \"c:/program files/gwb/gtdata/feoh.sdat\""):

    print "writing output to test_output.txt"
    myGWBrun.exec_cmd("show surfaces")
    print "Executing test"
    fp = open("test_results.txt", "w")
    if fp.closed:
```

```
            stderr.write("can't open test_results.txt")
        else:
            print "writing results to test_results.txt"
            cmds = ["pH = 8", "molality Cl- = .05", "go"]
            for cmd in cmds:
                myGWBrun.exec_cmd(cmd)

            pH = myGWBrun.results("pH")[0]
            fp.write("pH = %4.1f\n" % pH)

            Cl = myGWBrun.results("concentration Cl-")[0]
            fp.write("Cl = %12.5e molal\n" % Cl)

            Cl = myGWBrun.results("concentration Cl-", "mg/kg")[0]
            if Cl != ANULL:
                fp.write("Cl = %12.5e mg/kg\n\n" % Cl)
            else:
                fp.write("unit conversion failed - Cl = ANULL\n\n")

            Name = myGWBrun.results("species")
            Spec = myGWBrun.results("concentration aqueous")
            for i in range(len(Name)):
                fp.write("%-32s %12.5e molal\n" % (Name[i], Spec[i]))
            Spec = myGWBrun.results("concentration aqueous", "mg/kg")
            fp.write("\n\n")
            for i in range(len(Name)):
                fp.write("%-32s %12.5e mg/kg\n" % (Name[i], Spec[i]))
            fp.close()

    myGWBrun.destroy()
    raw_input("press return to exit> ")
```

6.5.5 Python command line

The GWB plug-in has been tested with Python for Windows version 2.7.3. The version of Python you are using must be the same as the version of GWB installed (32-bit vs. 64-bit).

To run the GWBplugin Example1 on the command line with Python, follow these steps:

```
# open the command prompt
cmd.exe

# create a working folder and change to that folder
mkdir "%homepath%\GWBplugin"
cd "%homepath%\GWBplugin"

# copy the "src" folder from GWB installation (default install path shown)
```

```
copy /Y "C:\Program Files\GWB\src"

# add the GWB installation folder to your path
set path=C:\Program Files\GWB;%path%

# run the example with Python
python GWBplugin_Python_Example1.py
```

6.6 MATLAB

GWB provides a GWBplugin wrapper class contained in the MATLAB script file "GWBplugin.m" which handles dealing with the C data type conversion and calling the DLL. In order to locate the GWB DLLs the GWBplugin class uses, you must add the GWB installation directory to the PATH environment variable.

Since MATLAB is a dynamically typed language, there are some minor differences with its "results" functions compared to statically typed languages.

To begin, locate the directory in which the GWB software is installed on your computer. Most commonly, the installation is in directory "C:\Program Files\Gwb" for 64 bit GWB, which we'll assume here, or in "C:\Program Files (x86)\Gwb" for the 32 bit version.

Add the GWB installation directory (e.g., "C:\Program Files\Gwb") to your PATH environmental variable, either from the Windows Control Panel before starting MATLAB, or by issuing the command

```
setenv('PATH',[getenv('PATH'),';C:\Program Files\GWB']);
```

from within MATLAB.

Next, set up a C compiler in MATLAB using the command "mex -setup", as described in the MATLAB documentation. The compiler might be cl, icl, or gcc; it should already have been installed on your computer.

Now, compile within MATLAB the file "GWBpluginMex.cpp" and associated header file "class_handle.hpp", which are located in the "src" subdirectory, to produce a MATLAB library. The command to do this is

```
mex "C:\Program Files\Gwb\src\GWBpluginMex.cpp"
        -I"C:\Program Files\Gwb\src"
        -L"C:\Program Files\Gwb" -lgwbplugin
```

6.6.1 GWBplugin MATLAB wrapper class overview

This is a synopsis of the MATLAB wrapper class provided in "GWBplugin.m", which can be found in the "src" directory of the GWB installation folder:

```
classdef GWBplugin < handle
    properties (SetAccess = private, Hidden = true)
        objectHandle;
    end
    methods
        function this = GWBplugin(varargin)
            this.objectHandle = GWBpluginMex('new');
            GWBpluginMex('initialize', this.objectHandle, varargin{:});
        end
```

```
        function delete(this)
            GWBpluginMex('delete', this.objectHandle);
        end

        function exec_cmd(this, varargin)
            GWBpluginMex('exec_cmd', this.objectHandle, varargin{:});
        end

        function result = results(this, varargin)
            result = GWBpluginMex('results', this.objectHandle, varargin{:});
        end
    end
end
```

6.6.2 Initializing the GWB application

Within your MATLAB script you begin by creating a "GWBplugin" object, passing the application name (e.g., 'spece8'), an optional file name for the GWB application to write output to, and any command-line type arguments.

```
myGWBrun = GWBplugin(app_name, file_name, cmds):
```

Parameters:

app_name

A string containing the GWB application name you wish to use. Valid options are rxn, spece8, react, x1t, and x2t.

file_name (optional)

A string containing the name of the file you want the GWB application to write its output to. Omit or pass an empty array if you do not want to write output to a file.

cmds (optional)

A string containing command-line options you could normally pass to the application when running it from the command-line. Can be omitted or an empty array.

Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value

The handle of the GWB plugin, or zero on failure.

Remarks

For this function to succeed you must have your GWB installation folder added to the PATH environment variable so all the required DLLs can be found.

Output to the file is not performed in MATLAB until the GWBplugin object has been cleared from memory. To do this, enter the MATLAB 'clear' command.

Examples

```
% plug-in SpecE8 with no output written and no options
myGWBrun = GWBplugin('spece8');
...

% plug-in React with output written to output.txt and no options
myGWBrun = GWBplugin('react','output.txt');
...

% plug-in X1t with no output written, no working directory change,
%  and read input from pb_contam.x1t
myGWBrun = GWBplugin('x1t',[ ],'-nocd
                        -i \"c:/Program Files/gwb/script/pb_contam.x1t\"');
```

6.6.3 Configuring and executing calculations

Use the "exec_cmd" function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a "go" command to trigger the calculations.

```
exec_cmd(myGWBrun, uline):
```

Parameter

uline

A string containing the command you wish to send to the GWB application.

Return value

Non-zero on success and zero on failure.

Remarks

You may include more than one GWB command in a single call.

Examples

```
exec_cmd(myGWBrun, '3 mmol H+')
exec_cmd(myGWBrun, '2 mmol/kg Ca++', '4 mmol/kg Cl-', 'go')
```

6.6.4 Retrieving the results

Transfer calculation results from the GWB application to your program with the "results" function. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual. Use the "results" functions by providing the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
results(myGWBrun, value, units, ix, jy):
```

Parameters:

value

String containing the report command keyword and arguments.

units (optional)

String containing the units you would like the results returned in. Omit or pass an empty array if you want default units.

ix (optional)

X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional)

Y node position. This is only used when running X2t, otherwise it is ignored.

Return value

Array containing the requested results.

Remarks

If you request a single value, it is returned as an array of length one.

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, an empty array is returned.

Parameter ix is used when running X1t and X2t; otherwise it is ignored. Paramter jy is similarly used only when running X2t.

Examples

```
Cl = results(myGWBrun,'concentration Cl-'); % in default units
fprintf('concentration of Cl- in molal is %10.4g\n',Cl);

Cl = results(myGWBrun,'concentration Cl-','mg/kg'); % in different units
fprintf('concentration of Cl- in mg/kg is %10.4g\n',Cl);

Name = results(myGWBrun,'species');
Spec = results(myGWBrun,'concentration aqueous','mg/kg');

fprintf('\n There are %i aqueous species\n\n',length(Name));
for i = 1:length(Name)
    fprintf('%-4s = %10.4g mg/kg\n',Namei,Spec(i));
end
```

6.6.5 Cleaning up

The "delete" function is designed to free up the underlying memory associated with the GWBplugin object. Due to a known issue in MATLAB, we recommend you reuse existing plugin instances, rather than destroy and recreate them.

To reuse an instance, issue the command

```
exec_cmd(myGWBrun, 'reset');
```

6.6.6 MATLAB code examples using the plug-in feature

Normally you would use the GWB plug-in within your program with no output being written to a file. The following is an example of this:

```
% Only needed if the GWB install directory is not in the PATH
% environment variable
setenv('PATH',[getenv('PATH'),';C:\Program Files\Gwb']);

% Create the plugin object and start the GWB program
myGWBrun = GWBplugin('spece8',[ ],'-nocd');

disp('Beginning run');
exec_cmd(myGWBrun,'pH = 8','molality Cl- = .05','go');
disp('Finished run');

% Ensure run was successful
if results(myGWBrun,'Success')
    %retrieve results
    pH = results(myGWBrun,'pH');

    Cl = results(myGWBrun,'concentration Cl-'); % in default units
    fprintf('concentration of Cl- in molal is %10.4g\n',Cl);
    Cl = results(myGWBrun,'concentration Cl-','mg/kg'); % in different units
    fprintf('concentration of Cl- in mg/kg is %10.4g\n',Cl);

    Name = results(myGWBrun,'species');
    Spec = results(myGWBrun,'concentration aqueous','mg/kg');

    fprintf('\n There are %i aqueous species\n\n',length(Name));
    for i = 1:length(Name)
        fprintf('%-4s = %10.4g mg/kg\n',Name{i},Spec(i));
    end
end
```

To familiarize yourself with the plug-in feature, you might want the GWB program's normal output and results to be written to the console and to text files. The following code shows examples of this:

```
% Only needed if the GWB install directory is not in the PATH
% environment variable
setenv('PATH',[getenv('PATH'),';C:\Program Files\Gwb']);
ANULL = -999999.0;

disp('Starting program SpecE8');
```

```
myGWBrun = GWBplugin('spece8','test_output.txt', ...
    '-nocd -i "C:/Program Files/Gwb/Script/Freshwater.sp8" ...
    -s "C:/Program Files/Gwb/Gtdata/FeOH.dat');

disp('writing output to test_output.txt');
exec_cmd(myGWBrun,'show surfaces'); % write to output file

disp('Executing test');

fp=fopen('test_results.txt', 'w');
if fp < 0
    disp('cant open test_results.txt');
else
    disp('writing results to test_results.txt');
    exec_cmd(myGWBrun,'pH = 8','molality Cl- = .05','go');

    pH = results(myGWBrun,'pH');
    fprintf(fp,'pH = %4.1f \n', pH);

    Cl = results(myGWBrun,'concentration Cl-');
    fprintf(fp,'Cl = %12.5e molal\n', Cl);

    Cl = results(myGWBrun,'concentration Cl-','mg/kg');
    if(Cl ~= ANULL)
        fprintf(fp,'Cl = %12.5e mg/kg\n\n', Cl);
    else
        fprintf(fp,'unit conversion failed - Cl = ANULL\n\n');
    end

    Name = results(myGWBrun,'species');
    Spec = results(myGWBrun,'concentration aqueous');

    for i = 1:length(Name)
        fprintf(fp, '%-32s %12.5e molal\n', Name{i}, Spec(i));
    end
    fprintf(fp,'\n\n');

    Spec = results(myGWBrun,'concentration aqueous','mg/kg');
    for i = 1:length(Name)
        fprintf(fp, '%-32s %12.5e molal\n', Name{i}, Spec(i));
    end

    fclose(fp);
end
```

6.6.7 MATLAB command line

The GWB plug-in has been tested with MATLAB versions 7.9 and 8.0. The version of MATLAB you are using must be the same as the version of GWB installed (32-bit vs. 64-bit).

To run the GWBplugin Example1 on the command line with MATLAB, follow these steps. First, after opening MATLAB, create a working folder and change to that folder

```
mkdir 'GWBplugin'
cd 'GWBplugin'
```

Copy the files “GWBplugin.m” and “GWBplugin_Matlab_example1.m” from the “src” folder of GWB installation into the new folder

```
copyfile ('C:\Program Files\GWB\src\GWBplugin.m', pwd)
copyfile ('C:\Program Files\GWB\src\GWBplugin_Matlab_example1.m', pwd)
```

Compile the MATLAB wrapper with the “mex” command

```
mex "C:\Program Files\Gwb\src\GWBpluginMex.cpp"
        -I"C:\Program Files\Gwb\src"
        -L"C:\Program Files\Gwb" -lgwbplugin
```

You are now ready to run the example script

```
GWBplugin_Matlab_example1
```

which should produce output similar to the following:

```
>>GWBplugin_Matlab_example1

Beginning run.
Finished run.

concentration of Cl- in molal is       0.05
concentration of Cl- in mg/kg is        1770

There are 4 aqueous species.

Cl-  =        1770 mg/kg
H+   =  1.139e-05 mg/kg
HCl  =  1.234e-11 mg/kg
OH-  =    0.02039 mg/kg
```

Follow the same procedure to run the second example script, “GWBplugin_Matlab_example2.m”. Congratulations on plugging into the GWB!

6.7 Other languages

Any language that can load DLLs, call C functions from them, and handle some basic C data types should be able to use the GWB plug-in feature. You must have your GWB installation folder added to the PATH environment variable so that all of the required DLLs can be found. The C data types that need to be handled are void*, char*, int*, double*, and int. If the language you want to use is similar to one that a wrapper is provided for, a good place to start is to look at how that wrapper is implemented.

To create a wrapper class, interface, or whatever makes sense for your target language, follow these steps:

- **Load the GWBplugin DLL**. Generally this will be done during run-time with a call to LoadLibrary or whatever the equivalent is in the language. Some languages, mostly compiled and linked ones, can instead link to the export library GWBplugin.lib.
- **Tell your program about the functions you will call from the DLL**. This is usually done by giving prototypes in some way or by directly including GWBplugin.h. The DLL functions and their prototypes are listed in the next section.
- **Encapsulate.** Create functions in your wrapper that call the corresponding DLL function and handle data type conversions. The wrapper, if possible, should also have a void* member variable that can be passed by address to the DLL functions. This void* member variable keeps track of a particular GWBplugin instance.

6.7.1 GWBplugin.dll function prototypes

Following is the list of the definitions and functions exported from GWBplugin.dll that your wrapper will need to use. Note that function parameters labeled as (optional) are in fact required when you call the C function. It is suggested, however, that you make these arguments optional for your own wrapper if possible and use the provided suggested defaults.

```
// GWBplugin.h

#define ANULL -999999.0              // marker for an undefined value

extern "C" _ _declspec(dllexport)
   int c_initialize(void* plugin, const char* app_name,
                    const char* file_name, const char* cmds);

extern "C" _ _declspec(dllexport)
   int c_exec_cmd(void* plugin, char* uline);

extern "C" _ _declspec(dllexport)
   int c_results(void* plugin, void* data, const char* value,
                    const char* units, int ix, int jy);
```

```
extern "C" _ _declspec(dllexport)
   int c_results_c(void* plugin,  void* data, const char* value,
                   const char* units, int ix, int jy, int* slen);

extern "C" _ _declspec(dllexport)
   int c_destroy(void* plugin);
```

6.7.2 Initializing the GWB application

Within your code, first create a void* equivalent variable or something that can hold a pointer data type... i.e. that is 32-bits long (for a 32-bit application) or 64-bits long (for a 64-bit application). This will be a member variable of your class if possible.

Next, use the “c_initialize” function to start the GWB application of interest by passing the address of the void* variable, the application name, an optional output file name, and any command-line type arguments. The “c_initialize” function must be called before calling any of the other functions.

```
int c_initialize (
   void* plugin,
   const char* app_name,
   const char* file_name,
   const char* cmds
);
```

Parameters:

plugin

A dereferenceable pointer that points to a pointer which can be assigned a value. It keeps track of a particular plugged-in GWB application.

app_name

String containing the GWB application name - “rxn”, “spece8”, “react”, “x1t”, or “x2t”.

file_name (optional) (default: NULL or empty string)

String containing the name of the file you want the GWB output written to. This can be NULL or an empty string if you do not want the output to be written to a file.

cmds (optional) (default: NULL or empty string)

String containing command-line options you could normally pass to the application when running it from the command-line. This can be NULL or an empty string.

Command-line options:

-cd	Change the working directory to the directory containing the input script specified with the -i option.
-nocd	Do not change the working directory.
-i <input_script>	Read initial input commands from the specified file.
-gtd <gtdata_dir>	Set directory to search for thermodynamic datasets.
-cond <cond_data>	Set the dataset for calculating electrical conductivity.
-d <thermo_data>	Set the thermodynamic dataset.
-s <surf_data>	Set a dataset of surface sorption reactions.
-iso <isotope_data>	Set a dataset of isotope fractionation factors.

Return value

Non-zero on success and zero on failure.

Examples

Some examples of how to start the GWB plug-in in various ways:

```
void* myPlugin = NULL;

// plug-in SpecE8 with no output written and no command-line options
int success = c_initialize(&myPlugin, "spece8");

// plug-in React with output written to output.txt and no command-line options
int success = c_initialize(&myPlugin, "react", "output.txt");

// plug-in X1t with no output written, no working directory change,
// and input read from pb_contam.x1t
int success = c_initialize(&myPlugin, "x1t", NULL, "-nocd
            -i \"c:/program files/gwb/script/pb_contam.x1t\"");
```

Function "c_destroy" can be used at the end of the program to free up the underlying memory associated with the plugged-in GWB application.

```
c_destroy(&myPlugin);
```

6.7.3 Configuring and executing calculations

Use the “c_exec_cmd” function to transmit commands to the GWB plug-in. Each application has a chapter in this reference manual that is a comprehensive guide to the commands available. Use these commands to configure the application and then send a “go” command to trigger the calculations.

```
int c_exec_cmd(
   void* plugin,
   char* uline
);
```

Parameters:
plugin
A dereferenceable pointer that has already been used with c_initialize. It keeps track of a particular plugged-in GWB application.
uline
String containing the command to be sent to the GWB application.
Return value
Non-zero on success and zero on failure.
Examples

```
c_exec_cmd(&myPlugin, "3 mmol H+");
c_exec_cmd(&myPlugin, "2 mmol Ca+");
c_exec_cmd(&myPlugin, "5 mmolar Cl-");
c_exec_cmd(&myPlugin, "go");
```

6.7.4 Retrieving the results

Transfer calculation results from the GWB application to your program with the “c_results” function. The keywords, arguments, default units, and return types are the same as those listed in the table in the **Report Command** chapter of this reference manual. Use the "c_results" function by providing the plugin parameter, the address of a data block to fill, the report command and keywords, optional desired units, and the node location of choice (X1t and X2t only).

```
int c_results(
    void* plugin,
    void* data,
    const char* value,
    const char* units,
    int ix,
    int jy
);
```

Parameters:
plugin
A dereferenceable pointer that has already been used with c_initialize. It keeps track of a particular plugged-in GWB application.
data
Address of data block to fill. This can be NULL to determine data block size.
value
String containing the report command keyword and arguments.
units (optional) (default: NULL or empty string)
String containing the units you want the results returned in. This can be NULL or an empty string if you want the results returned in the default units.
ix (optional) (default: 0)
X node position. This is only used when running X1t and X2t, otherwise it is ignored.

jy (optional) (default: 0)
Y node position. This is only used when running X2t, otherwise it is ignored.

Return value
The number of values written (or to be written) to the data block.

Remarks
To determine the size of data block you will need, first call this function with the data parameter as NULL and with the rest of the parameters filled. If you know that the report command you are using only returns a single value, you can simply pass a pointer to the correct data type. See the **Report Command** chapter for details on data types and available keywords.

If the command fails for any reason, for example if the requested data doesn't exist or the specified unit conversion failed, the data will be filled with ANULL (-999999.0). For this reason, you should "#define ANULL -999999.0" (or language equivalent) in your wrapper.

For languages that are dynamically typed (e.g. Python and Perl), you will either need to create multiple wrapper "results" functions (one for each possible data type: int, double, char*) or pass the expected type as an extra parameter. It is often best to omit the data parameter in the wrapper function. You then can call "c_results" with a NULL value for data to get the size, allocate C compatible memory, call "c_results" with the data parameter, convert data, and then return an array of the results. See GWBplugin.pm or GWBplugin.py for examples of this.

Examples

```
// get aqueous species names
int ndata = c_results(&myPlugin, NULL, "species");
char** Species = (char**) malloc(sizeof(char*) * ndata);
c_results(&myPlugin, Species, "species");

// get aqueous species concentrations in mg/kg
double* Conc = (double*) malloc(sizeof(double) * ndata);
c_results(&myPlugin, Conc, "concentration aqueous", "mg/kg");

// get pH at node 3,5
double pH = ANULL;
c_results(&myPlugin, &pH, "pH", NULL, 3, 5);
```

If you are retrieving string values and you need to know the string lengths for conversion purposes, you will need to use the "c_results_c" function. It is equivalent to the "c_results" function, but it also takes an extra parameter which will store the length of the strings.

```
int c_results_c(
   void* plugin,
   void* data,
   const char* value,
   const char* units,
```

```
    int ix,
    int jy,
    int* slen // address of data block to fill with retrieved string lengths
);
```

Examples

```
// get aqueous species names
int ndata = c_results(&myPlugin, NULL, "species");
char** Species = (char**) malloc(sizeof(char*) * ndata);
int* Lengths = (int*) malloc(sizeof(int) * ndata);
c_results_c(&myPlugin, Species, "species", NULL, 0, 0, Lengths);
```

Units Recognized

The following is a complete table of the unit names recognized by the GWB. The qualifier “free” specifies that the constraint applies to the free rather than to the bulk entry. Use the “log” qualifier to set the variable on a logarithmic scale. Examples:

```
Cl- 4.1 mg/kg
Cl- 4.1 free mg/kg
Cl- 0.612784 log free mg/kg
```

Dimension	Units			
Mass and Concentration	mol	mmol	umol	nmol
	molal	mmolal	umolal	nmolal
	mol/kg	mmol/kg	umol/kg	nmol/kg
	mol/l	mmol/l	umol/l	nmol/l
	kg	g	mg	ug
	ng			
	g/kg	mg/kg	ug/kg	ng/kg
	wt fraction	wt%		
	g/l	mg/l	ug/l	ng/l
	eq	meq	ueq	neq
	eq/kg	meq/kg	ueq/kg	neq/kg
	eq/l	meq/l	ueq/l	neq/l
	cm3	m3	km3	l
	mol/cm3	mmol/cm3	umol/cm3	nmol/cm3
	kg/cm3	g/cm3	mg/cm3	ug/cm3
	ng/cm3			
	mol/m3	mmol/m3	umol/m3	nmol/m3
	kg/m3	g/m3	mg/m3	ug/m3
	ng/m3			
	vol. fract.	volume%		

Dimension	Units			
Activity	activity	ratio		
Fugacity	fugacity			
Electrical Potential (Eh)	V	mV	pe	
pH	pH			
Percentage	%			
Time	s	min	hr	day
	mon	yr	m.y.	
Distance	mm	cm	m	km
	in	ft	mi	
Reaction Rate	mol/s	mmol/s	umol/s	nmol/s
	kg/s	g/s	mg/s	ug/s
	ng/s			
	cm3/s	m3/s	l/s	gal/s
	ft3/s			
	mol/min	mmol/min	umol/min	nmol/min
	kg/min	g/min	mg/min	ug/min
	ng/min			
	cm3/min	m3/min	l/min	gal/min
	ft3/min			
	mol/hr	mmol/hr	umol/hr	nmol/hr
	kg/hr	g/hr	mg/hr	ug/hr
	ng/hr			
	cm3/hr	m3/hr	l/hr	gal/hr
	ft3/hr			
	mol/day	mmol/day	umol/day	nmol/day
	kg/day	g/day	mg/day	ug/day
	ng/day			
	cm3/day	m3/day	l/day	gal/day
	ft3/day			
	mol/yr	mmol/yr	umol/yr	nmol/yr
	kg/yr	g/yr	mg/yr	ug/yr
	ng/yr			
	cm3/yr	m3/yr	l/yr	gal/yr
	ft3/yr			

Dimension	Units			
Reaction Rate	mol/m.y.	mmol/m.y.	umol/m.y.	nmol/m.y.
	kg/m.y.	g/m.y.	mg/m.y.	ug/m.y.
	ng/m.y.			
	cm3/m.y.	m3/m.y.	l/m.y.	gal/m.y.
	ft3/m.y.			
	mol/cm3/s	mmol/cm3/s	umol/cm3/s	nmol/cm3/s
	kg/cm3/s	g/cm3/s	mg/cm3/s	ug/cm3/s
	ng/cm3/s			
	cm3/cm3/s	volume%/s		
	mol/cm3/min	mmol/cm3/min	umol/cm3/min	nmol/cm3/min
	kg/cm3/min	g/cm3/min	mg/cm3/min	ug/cm3/min
	ng/cm3/min			
	cm3/cm3/min	volume%/min		
	mol/cm3/hr	mmol/cm3/hr	umol/cm3/hr	nmol/cm3/hr
	kg/cm3/hr	g/cm3/hr	mg/cm3/hr	ug/cm3/hr
	ng/cm3/hr			
	cm3/cm3/hr	volume%/hr		
	mol/cm3/day	mmol/cm3/day	umol/cm3/day	nmol/cm3/day
	kg/cm3/day	g/cm3/day	mg/cm3/day	ug/cm3/day
	ng/cm3/day			
	cm3/cm3/day	volume%/day		
	mol/cm3/yr	mmol/cm3/yr	umol/cm3/yr	nmol/cm3/yr
	kg/cm3/yr	g/cm3/yr	mg/cm3/yr	ug/cm3/yr
	ng/cm3/yr			
	cm3/cm3/yr	volume%/yr		
	mol/cm3/m.y.	mmol/cm3/m.y.	umol/cm3/m.y.	nmol/cm3/m.y.
	kg/cm3/m.y.	g/cm3/m.y.	mg/cm3/m.y.	ug/cm3/m.y.
	ng/cm3/m.y.			
	cm3/cm3/m.y.	volume%/m.y.		
	mol/m3/s	mmol/m3/s	umol/m3/s	nmol/m3/s
	kg/m3/s	g/m3/s	mg/m3/s	ug/m3/s
	ng/m3/s			
	m3/m3/s			
	mol/m3/min	mmol/m3/min	umol/m3/min	nmol/m3/min
	kg/m3/min	g/m3/min	mg/m3/min	ug/m3/min
	ng/m3/min			
	m3/m3/min			

Dimension	Units			
Reaction Rate	mol/m3/hr	mmol/m3/hr	umol/m3/hr	nmol/m3/hr
	kg/m3/hr	g/m3/hr	mg/m3/hr	ug/m3/hr
	ng/m3/hr			
	m3/m3/hr			
	mol/m3/day	mmol/m3/day	umol/m3/day	nmol/m3/day
	kg/m3/day	g/m3/day	mg/m3/day	ug/m3/day
	ng/m3/day			
	m3/m3/day			
	mol/m3/yr	mmol/m3/yr	umol/m3/yr	nmol/m3/yr
	kg/m3/yr	g/m3/yr	mg/m3/yr	ug/m3/yr
	ng/m3/yr			
	m3/m3/yr			
	mol/m3/m.y.	mmol/m3/m.y.	umol/m3/m.y.	nmol/m3/m.y.
	kg/m3/m.y.	g/m3/m.y.	mg/m3/m.y.	ug/m3/m.y.
	ng/m3/m.y.			
	m3/m3/m.y.			
	mol/kg/s	mmol/kg/s	umol/kg/s	nmol/kg/s
	g/kg/s	mg/kg/s	ug/kg/s	ng/kg/s
	cm3/kg/s			
	mol/kg/min	mmol/kg/min	umol/kg/min	nmol/kg/min
	g/kg/min	mg/kg/min	ug/kg/min	ng/kg/min
	cm3/kg/min			
	mol/kg/hr	mmol/kg/hr	umol/kg/hr	nmol/kg/hr
	g/kg/hr	mg/kg/hr	ug/kg/hr	ng/kg/hr
	cm3/kg/hr			
	mol/kg/day	mmol/kg/day	umol/kg/day	nmol/kg/day
	g/kg/day	mg/kg/day	ug/kg/day	ng/kg/day
	cm3/kg/day			
	mol/kg/yr	mmol/kg/yr	umol/kg/yr	nmol/kg/yr
	g/kg/yr	mg/kg/yr	ug/kg/yr	ng/kg/yr
	cm3/kg/yr			
	mol/kg/m.y.	mmol/kg/m.y.	umol/kg/m.y.	nmol/kg/m.y.
	g/kg/m.y.	mg/kg/m.y.	ug/kg/m.y.	ng/kg/m.y.
	cm3/kg/m.y.			
Flow Rate	cm3/s	m3/s	l/s	gal/s
	ft3/s			
	cm3/min	m3/min	l/min	gal/min
	ft3/min			

Dimension	Units			
Flow Rate	cm3/hr	m3/hr	l/hr	gal/hr
	ft3/hr			
	cm3/day	m3/day	l/day	gal/day
	ft3/day			
	cm3/yr	m3/yr	l/yr	gal/yr
	ft3/yr			
	cm3/m.y.	m3/m.y.	l/m.y.	gal/m.y.
	ft3/m.y.			
Velocity	mm/s	cm/s	m/s	km/s
	mm/hr	cm/hr	m/hr	km/hr
	mm/day	cm/day	m/day	km/day
	mm/mon	cm/mon	m/mon	km/mon
	mm/yr	cm/yr	m/yr	km/yr
	mm/m.y.	cm/m.y.	m/m.y.	km/m.y.
	in/s	ft/s	mi/s	
	in/hr	ft/hr	mi/hr	
	in/day	ft/day	mi/day	
	in/mon	ft/mon	mi/mon	
	in/yr	ft/yr	mi/yr	
	in/m.y.	ft/m.y.	mi/m.y.	
Specific Discharge	cm3/cm2/s	m3/m2/s	ft3/ft2/s	
	cm3/cm2/hr	m3/m2/hr	ft3/ft2/hr	
	cm3/cm2/day	m3/m2/day	ft3/ft2/day	
	cm3/cm2/mon	m3/m2/mon	ft3/ft2/mon	
	cm3/cm2/yr	m3/m2/yr	ft3/ft2/yr	
	cm3/cm2/m.y.	m3/m2/m.y.	ft3/ft2/m.y.	
Density	kg/cm3	g/cm3	mg/cm3	ug/cm3
	ng/cm3			
	kg/m3	g/m3	mg/m3	ug/m3
	ng/m3			
Titration Alkalinity	eq_acid	meq_acid	ueq_acid	neq_acid
	eq_acid/kg	meq_acid/kg	ueq_acid/kg	neq_acid/kg
	eq_acid/l	meq_acid/l	ueq_acid/l	neq_acid/l
	g/kg_as_CaCO3	mg/kg_as_CaCO3	ug/kg_as_CaCO3	ng/kg_as_CaCO3
	wt%_as_CaCO3			
	g/l_as_CaCO3	mg/l_as_CaCO3	ug/l_as_CaCO3	ng/l_as_CaCO3
	mmol/l_as_CaCO3			

Dimension	Units			
Titration Acidity	eq_base	meq_base	ueq_base	neq_base
	eq_base/kg	meq_base/kg	ueq_base/kg	neq_base/kg
	eq_base/l	meq_base/l	ueq_base/l	neq_base/l
Sorption Capacity	mol/grock	mmol/grock	umol/grock	nmol/grock
Exchange Capacity	eq/grock	meq/grock	ueq/grock	neq/grock
Pore Volumes	pore_volumes			
Dynamic Viscosity	cp	poise		
Pressure	Pa	MPa	atm	bar
	psi			
Permeability	m2	cm2	um2	
	darcy	mdarcy	udarcy	
Diffusion/Dispersion Coefficients	cm2/s	m2/s		
Distribution Coefficients (KDs)	l/kg	ml/g	ml/mg	
Activity Coefficients	act. coef.			
Electrical Conductivity	uS/cm	umho/cm		
Energy	J	kJ	cal	kcal
Energy Content	J/mol	kJ/mol	cal/mol	kcal/mol
Heat Capacity	J/g/C	J/kg/K	cal/g/C	
Thermal Conductivity	W/cm/C	W/m/K	cal/cm/s/C	cal/m/s/C
Internal Heat Source	J/cm3/s	J/cm3/yr	J/m3/s	J/m3/yr
	cal/cm3/s	cal/cm3/yr	cal/m3/s	cal/m3/yr
	W/cm3	W/m3		
Thermal Transmissivity	W/C	W/K	J/s/C	J/s/K
	cal/s/C	cal/s/K		
Percent Removal	% removal			
Saturation	Q/K			
Deuterium (^{2}H)	SMOW-2H			

Dimension	Units			
Tritium (^{3}H)	TU			
Foaming Agents	g/l	mg/l	ug/l	ng/l
Carbon 13	PDB			
Percent Modern Carbon	PMC			
Oxygen Demand	g/l	mg/l	ug/l	ng/l
Oxygen 18	SMOW-18O			
Sulfur 34	CDT			
Odor	TON			
Turbidity	NTU			
Corrosivity	Cor			
Colonies per Volume	colonies/ml			
Radioactive Emission per Volume	pCi/l			
Radioactive Exposure over Time	mrem/yr			
Temperature	C	F	K	R
Angle	radians	degrees		
Color	CU			
Number	number			
Text	text			

Graphics Output

Programs **Act2**, **Tact**, **Gtplot**, **P2plot**, and **Xtplot** can render plots on a variety of devices, including your computer screen and black-and-white and color printers. The programs can also save your plots in a variety of graphics formats; you can later import these images to documents, web pages, or presentations that you prepare with other software. Finally, you can copy your plots to the MS Windows clipboard and paste them directly into other applications, in a format meaningful to the application.

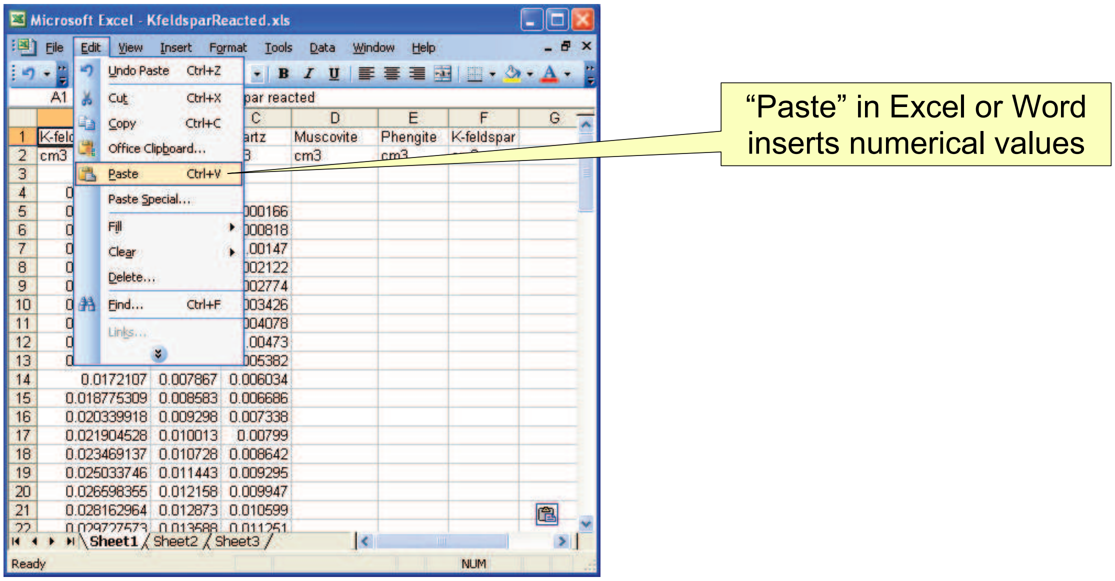
8.1 Clipboard

To copy the current plot to the clipboard, select **Edit** → **Copy** from the menubar on the graphics window, or touch **Ctrl+C**. You can then paste the plot directly into a variety of word processing and presentation graphics programs.

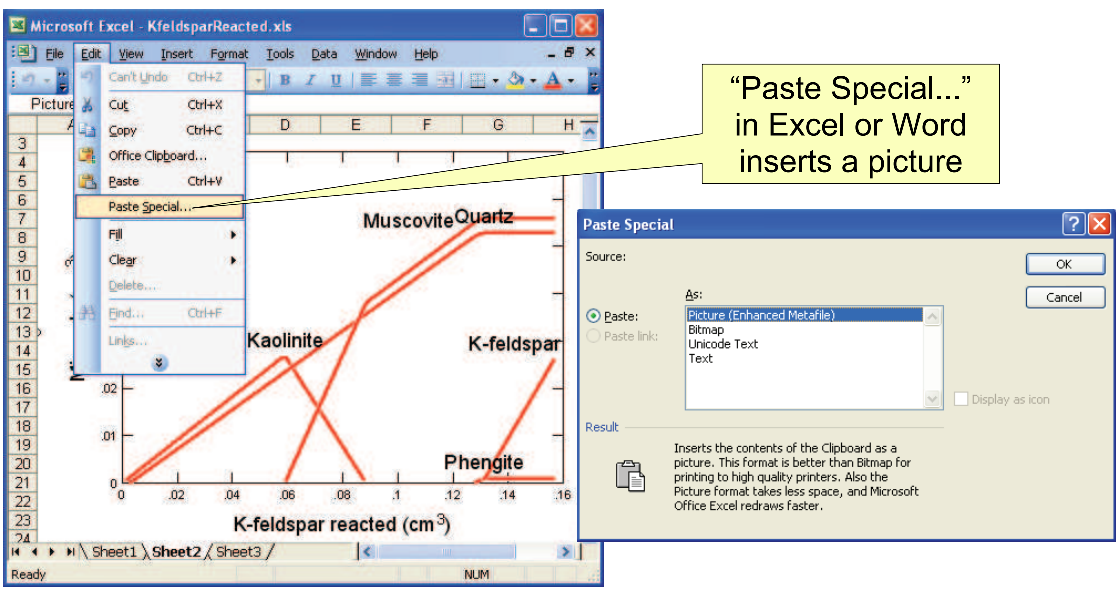
If you paste the plot into MS PowerPoint, it will appear as an EMF (an MS Enhanced Metafile) graphic object. Pasting into Adobe Illustrator places a native AI graphic.

If you paste a plot from **Gtplot** or **Xtplot** into MS Excel or a text editor such as Notepad or MS Word, the numerical values of the data points that make up the lines on the plot will appear in spreadsheet format.

You can control the format in which the plot is copied to the clipboard by selecting **Edit** → **Copy As**. You can choose to copy the plot as an AI object, an EMF object, a bitmap, or the data points in the plot, as tab delimited or space delimited text. Use the tab delimited option to paste the data into a spreadsheet program like MS Excel. For examining the data in a text file created with an editor like Notepad or MS Word, the space delimited option writes a nicely aligned table.



In MS Word or MS Excel, use **Paste Special. . .** to paste the plot as a picture instead.



8.2 Saving images

In many cases you will want to adjust label positions or change the annotation or coloring on your plot. Such changes can be made quickly using an illustration program such as PowerPoint. You can, furthermore, save images and import them into your reports or documents prepared with a word processor such as MS Word.

To save an image, select **File** → **Save Image. . .** from the menubar, then choose one of the file formats supported:

- PNG (.png)
- JPEG (.jpg)
- TIFF (.tif)
- Bitmap (.bmp)
- Enhanced Metafile (.emf)
- Adobe Illustrator (.ai)
- PDF File (.pdf)
- Scalable Vector Graphics (.svg)
- Compressed SVG (.svgz)
- Encapsulated PostScript (.eps)
- Color PostScript (.ps)
- Black-and-White PostScript (.ps)

Gtplot and **Xtplot** also support:

- Spreadsheet File (Tab delimited) (.txt)
- Text File (Space delimited) (.txt)

The programs save the plot images to files with names such as “Image_1.eps”, “Image_2.eps”, and so on; the suffix represents the file format (Encapsulated PostScript, in this case), as shown above.

Note that since each of these graphics formats has its own limitations, the plot once imported to another program may appear somewhat different than on your computer screen. Using your illustration program or word processor, however, you can quickly alter the diagram’s appearance to suit your needs.

When saving a PNG, JPEG, TIFF, or bitmap file, you may specify the quality of the saved image by choosing its resolution: High, Medium, Low, or Custom. Use **Custom. . .** to set the pixel width and height of the image, and to choose whether to preserve the aspect ratio of the plot.

Use the **Spreadsheet File (Tab delimited)** or **Text File (Space delimited)** option to save into a table the numerical coordinates of the data points on the plot. The spreadsheet table may be read directly into many popular spreadsheet programs.

Certain graphics types support font embedding. PDF files should always display and print properly, regardless of fonts installed on the system. PostScript files should also, if you have used the option to embed fonts. If you may want to edit the PostScript file, however, you should deselect the option to embed fonts, because programs such as Adobe Illustrator may restrict your ability to edit a document using embedded fonts (due to potential copyright/licensing issues). To edit these files, be sure that all of the required fonts are installed on your computer (see **Font for data markers** below).

When importing AI graphics to Adobe Illustrator, the program may prompt you to update the legacy text before you can edit the file. In this case, choose "Update". You need to release the clipping mask before you attempt to edit individual elements of the plot. Use the "Ungroup" and "Group" functions when repositioning or modifying elements.

8.3 Font for data markers

We supply a special TrueType font “GWBSymbol” to provide for data markers on scatter plots produced with **Act2**, **Tact**, **Gtplot**, **P2plot**, and **Xtplot**. The font is installed automatically on your computer when you install the GWB software.

The “GWBSymbol” font is not subject to copyright, so you can share it freely. If, for example, you send graphics output to a colleague, you can send her the font to install on her machine. In this way, the data markers on the plots will appear correctly.

Scatter Data

Act2, **Tact**, **Gtplot** and **Xtplot** can overlay data points (scatter data) on the diagrams that they produce. The preferred way to add scatter data is to import it from a **GSS** spreadsheet (".gss" file, see the **Using GSS** chapter of the **GWB Essentials Guide**). The old method of importing a specially formatted table from a text file still works, however.

To use this method to overlay a scatter plot of data points onto a diagram, first prepare a table of the data in a plain text (".txt") file created with an editor like Notepad.

The first line in the table is a header that names each column; e.g., Na+, pH, and so on. Enclose multiword names in quotes (e.g., "Mass solution", "Dissolved solids", etc.).

Subsequent lines contain the numeric data. A lack of data may be indicated by a string such as "n/d". To add error bars to the data points, enter a triplet of values separated by vertical bars (|). The values represent the minimum extreme of the error bar, the data point, and the maximum extreme. Exclude blank spaces from the triplet, or enclose it in quotes. The entry 0.5|2.0|3.5, for example, signifies a data point at 2.0 with an error bar extending from 0.5 to 3.5. The entry 0.5||3.5 prescribes the same error bar, omitting the symbol representing the data point.

The columns may be separated by any number of spaces or tabs. Comments may be placed anywhere in the table following a "#" sign. Comments extend from the # sign to the end of the line. You may also include, as separators in the table, blank lines or lines of dashes (-) underscores (_), or equal signs (=).

You may represent individual data points with a special symbol from among the choices in **Figure 12.1**, and choose the symbol's color and point size. To do so, append any or all of the following to the data line in question: the symbol name, its color, and its point size, using a string such as "12pt".

To load the dataset, select **File** → **Open** → **Scatter Data. . .** . Choosing the file selection dialog's **Edit** button allows you to modify the contents of the scatter data file. To see the changes on the diagram, use the **Open** button to reload the data file after saving the changes. You may clear the scatter data by choosing the **OFF** button.

■	square	□	box
●	blot	○	circle
▲	triangle	△	delta
▼	yield	▽	del
◆	diamond	◇	caution
•	bullet	∘	degree
★	star	☆	pentagram
⧗	hourglass	⧖	mobius
+	plus	×	cross

Figure 9.1 Symbols for plotting scatter data.

9.1 Act2 and Tact

To overlay a scatter plot of data points onto an **Act2** or **Tact** diagram, first prepare a table of the data in terms of log activities and fugacities. The first line identifies each of the table's columns in terms of a species or gas name, the ratio of two species or gases, or with the special labels "pH", "Eh", "pe" and "T(C)". The species or gas names should be those that appear on the diagram axes, or the original basis members of the thermodynamic dataset. When the axis variable is an activity ratio, the label should be formatted as numerator species^power/denominator species^power. A power of 1 does not need to be written explicitly. The activity ratio a_{Ca++}/a^2_{H+}, for example, should be written Ca++/H+^2.

Subsequent lines contain the numeric data in logarithmic form, except for the linear variables pH, Eh, pe, and temperature data. An example of a table dataset to be used with **Act2** is

```
pH    Na+   Ca++  HCO3-
------------------------------------------------------------
6.5   -4.3  -3.9  -3.3   red square 12pt
5.9   -3.2  -5.1  -4.0   # sample UI-4
6.8   -4.4  -3.6  n/d
```

Once the table is prepared, click on **File** → **Open** → **Scatter Data...** to select the dataset. The program will read the data and project them onto the diagram. The **Act2** and **Tact** command

```
scatter dataset_name
```

serves the same purpose.

9.2 Gtplot

Gtplot can overlay scatter data points on all of the plot types, except the pie and bar charts. To add scatter data to the "special" plots (ternary plot, Piper diagram, etc.), you specify the fluid composition, expressed in terms of thermodynamic components, as described below.

The first line in the table is a header that names each column; e.g., Temperature, pH, Carbon, Na+, and so on. Enclose multiword names in quotes (e.g., "Ionic strength", "Dissolved solids", etc.). You label the columns as follows:

- Components: Enter the component name (Ca++, SiO2(aq), etc.).
- Minerals: Enter the mineral name as it appears in the **Gtplot** menus (Quartz, Kaolinite, etc.).
- Species concentration: Enter the species name in parenthesis following the word "molality" (e.g., molality(Na+), molality(NaSO4-), etc.).
- Species activity: Enter the species name in parenthesis following the word "activity" (e.g., activity(Na+), activity(NaHCO3), etc.).
- Species activity coefficients: Enter the species name in parenthesis following the word "gamma" (gamma(Na+), gamma(NaCl), etc.).
- Elemental composition: Enter the element name (Oxygen, Carbon, etc.).
- Fugacities: Enter the name of the gas (CO2(g), Steam, etc.).
- Mineral saturation: Enter the mineral name in parenthesis following the word "logQoverK" (e.g., logQoverK(Albite), "logQoverK(Albite high)", etc.).
- Isotopic composition: Enter the fluid or mineral name in parenthesis following the isotope label (2-H, 18-O, etc.). Examples: "2–H(Bulk-system)", 18-O(CO2(aq)).
- Sorbed fractions: Enter the component name in parenthesis following the word "Xsorbed" (e.g., Xsorbed(Pb++)).
- Chemical parameters, Physical parameters, and Reactant properties: Enter the variable name as it appears in the **Gtplot** menus (Eh, "Dissolved solids", etc.).

The scatter dataset should contain numerical values for any of the following:

- The composition of the fluid, minerals, sorbate, and bulk system, expressed in terms of thermodynamic components. The components are the original basis entries in the thermo dataset, plus any decoupled redox species.
- One of the other variables (pH, Dissolved solids, Carbonate alkalinity, etc.) passed to **Gtplot** from **SpecE8** or **React**. These variables are those that appear in diagrams, or are used to calculate unit conversions.
- The masses of minerals in the modeled system.

- The concentrations, activities and activity coefficients of the dissolved species (Fe++, Na+, etc.).
- The elemental composition of the fluid, minerals, and bulk system.
- The fugacities of gases in the fluid.
- The saturation indices (log Q/K) for various minerals.
- The stable isotopic compositions of the fluid, minerals in bulk, individual minerals, and the entire system.
- The fractions of the various components sorbed onto mineral surfaces.

You enter the numerical values in terms of the following units:

- Enter component masses in mg/kg. In this case, concentrations are expressed in terms of the component species, such as HCO_3 or SO_4. A bicarbonate concentration, then, is entered directly in units of mg HCO_3/kg.
- Masses of minerals over the reaction path are entered in grams.
- The unit for species concentration is molality. The activities and the activity coefficients of dissolved species are unitless.
- Elemental compositions are entered in mg/kg. These values are expressed in terms of the elements, a convention not always followed when reporting anionic compositions. In creating a dataset, you need to convert analyses given in mg HCO_3/kg to mg C/kg, for example, SO_4 to sulfur, and so on.
- Gas fugacities are unitless.
- Mineral saturation is entered as the saturation index, log Q/K.
- Isotopic composition is entered in permil on the scale (e.g., SMOW) assumed in the **React** calculation.
- Sorbed fractions are entered as fractions.
- Values for the chemical and physical parameters and reactant properties are entered in their native units, which are the units shown when you first select a variable in the **XY Plot** dialog. The native unit for temperature, for example, is °C, ionic strength and Eh are given in molal and volts, and f_{O2} is entered as log fugacity. The native units for Mass solution and Mass H2O are kg; those for Carbonate alkalinity are mg/kg as $CaCO_3$.
- A lack of data may be indicated by a string such as “n/d”.

An example of a table dataset to be used with **SpecE8** is

```
pH     Na+    Fe++   "Dissolved solids"
--------------------------------------------------------
```

```
3.3    .15     5      1000
2.6    .12    15       800
6.     .5     n/d      200  # Buffalo Bayou
4.3    .75    0.1      450
```

To render these data, run **SpecE8** on a problem containing Na^{+} and Fe^{++} to produce a "SpecE8_plot.gtp" file, start **Gtplot**, and load the scatter dataset with **File** → **Open** → **Scatter Data. . . .** From the **Plot** menu, choose the **XY Plot**, for example, and set Na^{+} and Fe^{++} on the axes.

Now, if you choose an appropriate unit for the axis (μg/kg, mg/kg, g/kg, or wt%), the data points for the Na^{+} and Fe^{++} columns appear on the plot. You may choose another unit, such as molal concentration or grams, but you must first include in the scatter dataset columns for Mass H2O and Mass solution so that **Gtplot** can convert the data from mg/kg to the plotting unit; if you do not do so, **Gtplot** assumes a value of 1 kg for both variables.

In plots showing the variable "HCO_3 + CO_3", you should enter a value in the scatter dataset for Carbonate alkalinity, in mg/kg as $CaCO_3$, so that **Gtplot** can render the variable correctly. If you do not enter a value for alkalinity, the program will render the variable in terms of the reported HCO_3 concentration, if available, in which case the diagram produced will not account for the speciation of carbonate.

To assign symbols to each column of variables on an xy plot, add a line to the table with symbol names (or a place holder such as "--"). Set symbol color and point size in a similar manner. For example, adding the lines

```
--    --   circle   square
--    --   red      green
--    --   24pt     30pt
```

to the example above causes Fe^{++} concentration to be represented by 24pt red circles, and Dissolved solids by 30pt green squares. Choose from among colors shown on the **Choose Color. . .** dialog, enclosing multiword colors in quotes (e.g., "Cornflower blue"). To set marker point size, use fields like 15pt, 30pt, and so on.

To assign symbols to each sample (i.e., each line in the dataset), append any or all of the following to the data line in question: the symbol name, its color, and its point size. The symbol specifications on the sample line will be used to represent each sample in all of the special plots. On an xy plot, you may choose whether to assign the symbol, its color, and its size according to the variable (by analyte) or the sample (by sample). For example, the table

```
pH     Na+    Fe++   "Dissolved solids"
---------------------------------------------------------------------------
3.3    .15     5      1000         circle     24pt   red
2.6    .12    15       800         box        24pt   green
6.     .5     n/d      200         delta      24pt   blue
4.3    .75    0.1      450         caution    24pt   yellow
```

--	--	circle	square
--	--	red	green
--	--	24pt	30pt

would allow great flexibility in how to depict the data on an xy plot.

An example of a table dataset to be used with **React** is

pH	Eh	Iron	Sulfur	
3.3	.15	5	100	
2.6	.12	15	80	
6.	.5	n/d	20	# Buffalo Bayou
4.3	.75	0.1	45	

To render these data, run **React** on a problem containing iron and sulfur to produce a "React_plot.gtp" file, start **Gtplot**, and load the scatter dataset with the **File** → **Open** → **Scatter Data...** option. From **Plot** → **XY Plot...** choose to plot the elemental composition of the fluid. (Similarly, if you had entered data in columns labeled Fe++ and SO4–, you would choose to plot components in the fluid.) Select either pH or Eh as the variable for the x axis, and iron, sulfur, or both, for the y axis.

Now, if you choose an appropriate unit for the y axis (μg/kg, mg/kg, g/kg, or wt%), the data points for the iron and sulfur columns appear plotted against pH or Eh. You may choose another unit, such as molal concentration or grams, but you must first include in the scatter dataset columns for Mass H2O and Mass solution so that **Gtplot** can convert the data from mg/kg to the plotting unit; if you do not do so, **Gtplot** assumes a value of 1 kg for both variables.

In "special" plots (Piper diagrams, etc.) showing the variable "HCO_3 + CO_3", you should enter a value in the scatter dataset for Carbonate alkalinity, in mg/kg as $CaCO_3$, so that **Gtplot** can render the variable correctly. If you do not enter a value for alkalinity, the program will render the variable in terms of the reported HCO_3 concentration, if available, in which case the diagram produced will not account for the speciation of carbonate.

In order to see TDS represented as circles on Piper, ternary, and Durov diagrams, you should enter a value for "Dissolved solids" for each sample.

In a similar fashion, you could plot Eh against pH by choosing the "Chemical parameters" option from **Plot** → **XY Plot...**. If scatter values lie outside the plot axis ranges, touch **Ctrl+Z**.

A second example of a scatter dataset to be used with **React** is

"Rxn progress"	activity(NaCl)	"logQoverK(Albite low)"
.3	5.13e-9	.2
.5	5.15e-9	.1
.7	5.17e-9	.15
.9	5.19e-9	n/d

To render these data, run **React** on a problem containing sodium and silicon to produce a "React_plot.gtp" file, start **Gtplot**, and load the scatter dataset with the **File** → **Open** → **Scatter Data. . .** option. From **Plot** → **XY Plot. . .** choose to plot "Species activities". Select "Reaction Progress" as the variable for the x axis, and "NaCl" for the y axis. Choose to plot "Mineral Saturation" to show the scatter data for saturation index of the mineral "Albite low".

9.3 Xtplot

Xtplot can overlay scatter data points on all of the plot types, except the pie and bar charts. The first line in the table is a header that names each column. Examples: Length, Temperature, Carbon, Na+, etc. Enclose multiword names in quotes (e.g., "Ionic strength", "Dissolved solids", etc).

Label the column as follows:

- Components: Enter the component name (Ca++, SiO2(aq), etc.).
- Minerals: Enter the mineral name as it appears in the **Xtplot** menus (Quartz, Kaolinite, etc.).
- Species Concentration: Enter the species name in parenthesis following the word "molality" (molality(Na+), molality(NaSO4-), etc.).
- Species Activity: Enter the species name in parenthesis following the word "activity" (activity(Na+), activity(NaHCO3), etc.).
- Species Activity Coefficients: Enter the species name in parenthesis following the word "gamma" (gamma(Na+), gamma(NaCl), etc.).
- Elemental Composition: Enter the element name (Oxygen, Carbon, etc.).
- Fugacities: Enter the name of the gas (CO2(g), Steam, etc.).
- Mineral Saturation: Enter the mineral name in parenthesis following the word "logQoverK" (logQoverK(Albite), "logQoverK(Albite high)", etc.).
- Sorbed Fractions: Enter the component name in parenthesis following the word "Xsorbed" (e.g., Xsorbed(Pb++)).
- Chemical parameters, Physical parameters, and Reactant properties: Enter the variable name as it appears in the **Xtplot** menus (Eh, "Dissolved solids", etc.).
- Label position and time as Length, Width, or Time.

An example of a table dataset to be used with **Xtplot** is

```
Time   pH     Zinc   Sulfur
---------------------------------------------------------------------------
.15    3.3     5      100
.30    2.6    15       80
```

```
.50    6.     n/d    20    # Sample 98-010
.55    4.3    0.1    45
```

To render these data, run **X1t** or **X2t** on a problem containing zinc and sulfur to produce an “X1t_plot.xtp” or “X2t_plot.xtp” file. Start **Xtplot** and load the scatter dataset with **File** → **Open** → **Scatter Data. . . .** From **Plot** → **XY Plot. . .** choose to plot the elemental composition of the fluid versus time. (Similarly, if you had entered data in columns labeled “Fe++” and “SO4–”, you would choose to plot components in the fluid.) Select zinc, sulfur, or both, as variables to plot on the y axis.

Now, if you choose an appropriate unit for the y axis (μg/kg, mg/kg, g/kg, or wt%), the data points for the zinc and sulfur columns appear plotted against time. You may choose another unit, such as molal concentration or grams, but you must first include in the scatter dataset columns for "Mass H2O" and "Mass solution" so that **Xtplot** can convert the data from mg/kg to the plotting unit.

In “special” plots (Piper diagrams, etc.) showing the variable “HCO_3 + CO_3”, you should enter a value in the scatter dataset for Carbonate alkalinity, in mg/kg as $CaCO_3$, so that **Xtplot** can render the variable correctly. If you do not enter a value for alkalinity, the program will render the variable in terms of the reported HCO_3 concentration, if available, in which case the diagram produced will not account for the speciation of carbonate.

Alternatively, choose to plot “Chemical parameters” and set pH as the y axis variable. In this case, the scatter values for the column labeled pH appear on the plot. If scatter values lie outside the plot axis ranges, touch **Ctrl+Z**.

A second example of a scatter dataset to be used with **Xtplot** is

```
Length    Width    activity(NaOH)    logQoverK(NaBr)
--------------------------------------------------------------------------
 2e4       2e4       5e-12            -12
 4e4       4e4       6e-12            -10
 6e4       6e4       7e-12            -6
 8e4       8e4       8e-12            n/d
```

To render these data, run **X1t** or **X2t** on a problem containing sodium to produce a “X1t_plot.xtp” or “X2t_plot.xtp” file, start **Xtplot**, and load the dataset with **File** → **Open** → **Scatter Data. . . .** From **Plot** → **XY Plot. . .** choose to plot “Species activities” and set “NaOH” as the y-axis variable. Select “X position” as the variable for the x axis. Choose to plot “Mineral Saturation” to show the scatter data for saturation index of “NaBr”. You must include both “Length” (X position) and “Width” (Y position) values to show the scatter data on a mapview plot.

You can choose to represent each sample (i.e., each line in the dataset) with a user defined symbol. To assign symbols, append the symbol name to the line in question. In a similar fashion, you can assign the symbol’s color and point size. For example, if you enter a line

```
210 12.5 110 red 24pt square
```

the corresponding data point will appear as a red, 24 point square. Choose from among colors shown on the **Choose Color...** dialog, enclosing multiword colors in quotes (e.g., "Cornflower blue"). To set marker point size, use fields like 15pt, 30pt, and so on.

You can further assign a default symbol to each column whose variable is plotted along the y axis, as well as the symbol's default color and point size. To do so, add a line to the table with symbol names (or a place holder such as "--"), symbol colors, or symbol sizes. Adding the lines

```
--   --   box    cross
--   --   blue   red
--   --   12pt   16pt
```

to the first example in this section causes zinc concentration to be represented on a plot of elemental composition by 12 point blue boxes, and sulfur by 16 pt red crosses. The symbol specifications on the sample line will be used to represent each sample in all of the special plots. On an xy plot, you may choose whether to assign the symbol, its color, and its size according to the variable (by analyte) or the sample (by sample).

Multiple Analyses

It is not uncommon to have stored in a spreadsheet the results of a number of chemical analyses that you would like to enter – one at a time – into one of the GWB applications. You might wish to use **SpecE8**, for example, to figure calcite saturation or CO_2 fugacity for a group of analyses.

You can store multiple analyses in a **GSS** spreadsheet, select one or more samples, then launch **SpecE8** or **React** with **Analysis** → **Launch. . .** . An instance will be launched configured with the values in the first sample. Alternatively, you can add calculated analytes for all of your analyses to your **GSS** spreadsheet directly. See "Calculating analytes" in the **Using GSS** chapter of the **GWB Essentials Guide**.

If you have relatively few analyses in another type of spreadsheet, you may use the GWB's "drag and drop" feature. Highlight data for each analysis in the spreadsheet, left-click, drag into the GWB app, and calculate the desired result. For details, refer to the **Importing data** section of the **Introduction** to the **GWB Essentials Guide**.

Given a large number of chemical analyses, this procedure becomes tedious. It is best in this case to prepare a short script that performs the operations automatically, adding the results to the spreadsheet. This chapter describes how to do so.

You may also want to take advantage of the "scatter data" feature of the GWB, which allows chemical analyses to be overlain as data points on diagrams produced by **Act2**, **Tact**, **Gtplot**, and **Xtplot**. For more information, refer to the **Scatter data** sections of the corresponding chapter (**Using Act2**, and so on) in the GWB documentation set.

10.1 Calculation procedure

Suppose you have a number of chemical analyses stored in an Excel spreadsheet, and you would like to add to the spreadsheet results calculated by one of the GWB applications. To do so, follow this procedure:

- Save the spreadsheet from Excel as a tab-delimited text file. Go to **File** → **Save As. . .** and choose "Text (Tab delimited) (*.txt)" or "Unicode Text (*.txt)" as the file type. Excel will create a new file with a ".txt" file extension.
- Prepare and run a GWB script, such as the one in the next section, that runs within the GWB application. The script takes the text file as input and produces

a new text file containing the original data as well as the calculation results. The **Control Scripts** chapter in this **GWB Reference Manual** describes how to prepare GWB scripts. An example of such a script, which you may take and modify for your purposes, is given in the next section.

- Open (**File** → **Open. . .** ; choose “All Files (*.*) as the file type) the resulting text file in Excel. You can now save this file as an Excel spreadsheet (a “.xls” file).

The next section carries you through an example of this procedure.

10.2 Example calculation

The files installed under directory “Scripts\Spreadsheet” within the GWB installation directory (e.g., “\Program Files\GWB”) provide an example of using a script to process multiple analyses from an Excel spreadsheet. To run the example, copy files “Spreadsheet.xls” and “Script.xls” from this directory to a convenient location on your computer, such as the “My Documents” folder.

The analyses are stored in file “Spreadsheet.xls”. Open this file in Excel by double-clicking on it. Save it as a tab-delimited text file, as described in the previous section. This creates a file “Spreadsheet.txt”.

You may examine this file with an editor such as Notepad. It looks like

```
ID        pH    HCO3-  SO4--  Cl-   Ca++   Mg++  Na+
GW-12     6.78  585.7  309    56    205.6  63.9  21.4
GW-13     6.78  585.7  311    56.2  214.9  66.8  22.6
GW-14     6.85  652.8  582    42.6  269.2  89    25.8
GW-15     7     558.2  400    65.4  216.2  65.7  32
(and so on)
```

The first line in the file contains column headers including “pH” and various basis species, and subsequent lines contain the numerical data. The headers will be used together with the numerical values to create SpecE8 commands such as

```
pH = 6.78
HCO3- = 585.7 mg/kg
```

File “Script.sp8” contains a **SpecE8** script that reads “Spreadsheet.txt”, calculates CO_2 fugacity and calcite saturation, and writes a file “Output.txt”. You can modify this script for your own purposes.

The script is shown below. For clarity, **SpecE8** commands within the script are listed in bold face and comment lines are in italics; the actual file, of course, is simply a text file.

```
script start
# Set up the input and output.
set in_id [open "Spreadsheet.txt" r]
set out_id [open "Output.txt" w]
```

```
fconfigure $out_id -encoding unicode

# First line contains column headers; check for Unicode.
gets $in_id headers
if {![string is ascii $headers]} {
   close $in_id
   set in_id  [open "Spreadsheet.txt " r]
   fconfigure $in_id -encoding unicode
   gets $in_id headers
}
puts $out_id "$headers\tf CO2\tCalcite SI"

# Loop through remaining lines.
gets $in_id aline
report set_digits 4
while {$aline != ""} { set i 0
   reset; balance on Cl-

   # Set basis constraints from input data.
   foreach a [lrange $aline 1 end] {
      incr i 1
      if {[lindex $headers $i] == "pH"} {
         pH = $a
      } else {
         $a = [lindex $headers $i] mg/kg
      }
   }

   # Run SpecE8 calculation and write data + results.
   go
   foreach a [lrange $aline 0 end] {
      puts -nonewline $out_id "$a\t"
   }
   if {[report success]} {
      puts $out_id \
           "[report fugacity CO2(g)]\t[report SI Calcite]"
   } else {
      puts $out_id "Did not converge"
   }

   # Next line of input.
   gets $in_id aline
}
# Clean up.

close $out_id
```

```
close $in_id
quit
```

Double click on file “Script.sp8” to start **SpecE8** and execute the script. The program will produce a file “Output.txt” that contains the original data with the values calculated for CO_2 fugacity and calcite saturation appended as new columns. The file looks like

D	pH	HCO3-	SO4--	...	Na+	f CO2	Calcite SI
GW-12	6.78	585.7	309		21.4	0.06537	0.2281
GW-13	6.78	585.7	311		22.6	0.06515	0.2427
GW-14	6.85	652.8	582		25.8	Did not converge	
GW-15	7	558.2	400		32	0.04134	0.4714

Open “Output.txt” in Excel by selecting **File** → **Open...** and choosing “All Files (*.*)” for the file type. Follow the Excel “Text Import Wizard”, accepting the default at each step: “Delimited” file type, “Tab” delimiter, “General” data format. The calculation results will appear as would any spreadsheet, which you may save as an Excel (“.xls”) file.

Remote Control

You can run various GWB application programs not only by hand from the keyboard, but by "remote control" from a program or script you write. Note that this is now a legacy feature that has been replaced by the **Plug-in Feature**, and is no longer supported. **Rxn**, **Act2**, **Tact**, **SpecE8**, **React**, **X1t**, and **X2t** can be run in this way. The program you write serves as the "master program", which controls the GWB application as a "slave program".

In writing a program of your own, for example, you might need to determine the saturation state of calcite in a fluid of arbitrary composition. Instead of developing code to calculate the distribution of mass and mineral saturation states in a fluid, you could invoke **SpecE8** from within your program and let it do the work for you.

Similarly, you could use the remote control feature to balance reactions with **Rxn**, calculate activity diagrams with **Act2**, or figure the results of irreversible reaction paths with **React**. In each case, you configure the GWB application by sending text commands, trigger the calculation, and then retrieve the calculation results to use for your own purposes.

You can transfer the results from the slave application to the master program with the "report" command, as described in the **Report Command** chapter in this **GWB Reference Manual**. Or, as is especially useful with **Act2** and **Tact**, you can copy calculation results such as activity diagrams to your computer's clipboard, where they can be retrieved as graphical images. To do so, you use the "clipboard" command. Finally, you can simply read datasets, such as "SpecE8_output.txt" produced by the GWB applications, into the master program.

Your program, the master program, controls a GWB application as a slave program through a interprocess communications device known as a "pipe". (Pipes are not available in MS Windows 98 or ME, so you cannot use the remote control feature under these operating systems.) There are two ways to set up the communications. You can create two "unnamed pipes", one for input to and the other for output from the GWB application. Or, you can establish a "named pipe", which allows bidirectional data transfer.

Using a named pipe has a couple of advantages over unnamed pipes. First, the master program's standard input and output streams are available for use in the normal manner. Second, by establishing two or more pipes with different names, any number of copies of the GWB application programs can be invoked simultaneously.

Sending data through a pipe is much like writing to a file, and receiving data is like reading from a file. Running a GWB application by remote control, therefore, involves little more than standard programming techniques already familiar to anyone with modest programming experience.

To run a GWB application by remote control, you start it from the master program using the "-pipe" command line option. This option is followed by the name of the pipe, or for an unnamed pipe the keyword "stdio". In MS Windows, pipes are located in a pseudo-directory at the top level of the file system called "pipe".

If the master program, for example, has created an unnamed pipe, it could invoke program **SpecE8** using the command

```
spece8 -pipe stdio
```

In this case, the standard output stream of the master serves as standard input to the slave, and the slave's standard output stream is the master's input stream.

Similarly, if the master program has created a named pipe called "\pipe\mypipe", it could invoke program **SpecE8** by using the command

```
spece8 -pipe \pipe\mypipe
```

The master program could then communicate with **SpecE8** by writing to and reading from the pipe.

The sections below show examples of how the remote control feature can be implemented in the C++ programming and Tcl scripting languages, using named and unnamed pipes.

11.1 C++ program using unnamed pipes

In writing a program in C++, you will likely find it easiest to use a set of helper functions contained in file "RC_helper.cpp", a copy of which is installed in the "src" subdirectory of the GWB installation directory (e.g., in "\Program Files\GWB\src"). The helper functions in this file include:

`OpenGwbApplic`	Start the GWB program of interest.
`SendCommand`	Transmit a command to the GWB app, and, optionally, receive the results of the command.

There is a version of each function for unnamed and named pipes. By using these functions, the programmer can avoid worrying about the details of communication between the master and slave programs.

If you #include the header file "RC_helper.h" at the top of your master program, the helper functions will be available. Of course, you can modify and extend these functions for your own purposes, if you wish. The program must also be compiled with the "RC_helper.cpp" file, also provided in the same location.

In the following example, included in the "src" subdirectory, a console program invokes **React** using unnamed pipes to integrate a kinetic rate law for quartz dissolution at 100°C. Array “script” is a vector of pointers to the commands needed to configure **React** and trigger the calculation.

The program opens **React**, sends it the commands in array “script”, uses the “report” command to request the calculation result, which it extracts from **React’s** response using “sscanf”. Note that since the program uses unnamed pipes, output to the console is sent via the “stderr” output stream.

```
/* RC_example1.cpp */

#include "/Program Files/GWB/src/RC_helper.h"
#include <stdio.h>

char* script[] = {
   "reset",
   "time begin = 0 days, end = 5 days",
   "T = 100",
   "SiO2(aq) = 1 umolal",
   "react 5000 g Quartz",
   "kinetic Quartz rate_con = 2.e-15 surface = 1000",
   "go",
   ""
};

int main(int argc, char* argv[])
{
   char line[200];
   char discard[20];
   char** command;
   double SI_Quartz;

   fprintf(stderr, "Starting program React.\n");
   OpenGwbApplic("\\Program Files\\gwb\\react.exe");

   for (command = script; **command; command++)
      SendCommand(*command);

   SendCommand("report SI Quartz", line, sizeof(line));
   sscanf(line, "%lg", &SI_Quartz);
   fprintf(stderr, "Value of SI Quartz is %g.\n", SI_Quartz);

   SendCommand("quit");

   fprintf(stderr, "press return to exit> ");
   gets_s(discard);
   return 0;
}
```

11.2 C++ program using named pipes

As a second example of a master program written in C++, we open two copies of **React** as slave programs; the copies run simultaneously. To do so, we establish two pipes, using the “Pipe” class defined in “RC_helper.h”. In this case, the standard I/O channels are available to the program, so we need not direct console messages to “stderr”.

```
/* RC_example2.cpp */

#include "/Program Files/GWB/src/RC_helper.h"
#include <stdio.h>

char* script1[] = {
   "reset",
   "time begin = 0 days, end = 5 days",
   "T = 100",
   "SiO2(aq) = 1 umolal",
   "react 5000 g Quartz",
   "kinetic Quartz rate_con = 2.e-15 surface = 1000",
   "go",
   ""
};

char* script2[] = {
   "reset",
   "time begin = 0 days, end = 5 days",
   "T = 100",
   "SiO2(aq) = 1 umolal",
   "react 5000 g Quartz",
   "kinetic Quartz rate_con = 2.e-15 surface = 750",
   "go",
   ""
};

int main(int argc, char* argv[])
{
   char line[200];
   char discard[20];
   char** command1;
   char** command2;
   double SI_Quartz;
   Pipe pipe1("pipe1");
   Pipe pipe2("pipe2");

   printf("Open two copies of React.\n");
   OpenGwbApplic(pipe1,
```

```
                "\\Program Files\\gwb\\react.exe");
   OpenGwbApplic(pipe2,
                "\\Program Files\\gwb\\react.exe");

   for (command1 = script1, command2 = script2;
        **command1 || **command2;
        **command1 ? command1++ : 0, **command2 ? command2++ : 0) {
      if (**command1)
         SendCommand(pipe1, *command1);
      if (**command2)
         SendCommand(pipe2, *command2);
   }

   SendCommand(pipe1, "report SI Quartz", line, sizeof(line));
   sscanf(line, "%lg", &SI_Quartz);
   printf("SI Quartz for 1000 cm2/g is %g.\n", SI_Quartz);

   SendCommand(pipe2, "report SI Quartz", line, sizeof(line));
   sscanf(line, "%lg", &SI_Quartz);
   printf("SI Quartz for 7500 cm2/g is %g.\n ", SI_Quartz);

   SendCommand(pipe1, "quit");
   SendCommand(pipe2, "quit");

   printf("press return to exit> ");
   gets_s(discard);
   return 0;
}
```

11.3 Tcl script using unnamed pipes

You may find it especially useful to invoke GWB applications from within another application or calculation environment, such as a spreadsheet, word processor, or mathematical interpreter. You can do so, as long as the environment has scripting abilities and can open pipes.

As an example, we repeat the first example above in the Tcl scripting language. As in the C++ example, a number of helper functions are available in file “RC_helper.tcl”, installed with the GWB in subdirectory src. The complete Tcl script is given below.

```
source RC_helper.tcl

set cmdlist {
   reset
   {time begin = 0 days, end = 5 days}
   {T = 100}
   {SiO2(aq) = 1 umolal}
```

```
   {react 5000 g Quartz}
   {kinetic Quartz rate_con = 2.e-15 surface = 750}
   go
}

OpenGwbApplic {/Program Files/gwb/react/react.exe}
foreach cmd $cmdlist {
   SendCommand $cmd
}
SendCommand {report SI Quartz} line
puts "SI Quartz is $line."
SendCommand quit
```

11.4 Perl script using unnamed pipes

As a final example, we show how to run **React** by remote control from a Perl script. The example below uses the object oriented Perl module “RC_helper.pm”, included in the “src” subdirectory of the GWB installation.

```
#!/usr/bin/env perl
use strict;
use warnings;
use lib "\\Program Files\\Gwb\\src";
use RC_helper;

my $script = <<SCRIPT;
reset
time begin = 0 days, end = 5 days
T = 100
SiO2(aq) = 1 umolal
react 5000 g Quartz
kinetic Quartz rate_con = 2.e-15 surface = 1000
go
SCRIPT

print "Starting program React.\n";
my $react = RC_helper->new("\\Program Files\\Gwb\\react.exe");
for my $command (split /\n/, $script) {
   $react->send_command($command);
}
my $SI_Quartz = $react->send_command("report SI Quartz");
print "Value of SI Quartz is $SI_Quartz\n";

$react->send_command("quit");
```

Index