HumMod 2010 Schema

HumMod

About

HumMod Schema

Tom Coleman
support@biosim.com

Robert Hester
rhester@physiology.umsmed.edu

Richard Summers
rsummers@emergmed.umsmed.edu

Department of Emergency Medicine

Department of Physiology & Biophysics

University of Mississippi Medical Center

2500 N. State St.

Jackson, MS 39216 USA

physiology.umc.edu/themodelingworkshop/

http://groups.google.com/group/modelingworkshop

Preface

These documents describe the rules used by HumMod, and possibly other equation solvers, to define a mathematical model.

Our intent was that the simulation environment would not be associated with proprietary hardware, software and formatting. Extensible Markup Language (XML) was selected as the grammar.

Briefly, XML uses names in angle brackets (< and >) to identify data. The brackets and names are called *tags*. Data lies between two tags. The tag – data – tag sequence is called an *element* and the data is called the element *content*.

There are five volumes in this document. The main task of these volumes is to describe all of the XML elements that we have defined for out modeling environment.

Volume 1. Model This volume develops three topics. The first introduces XML. The second describes some additional grammatical rules to be used in model definition. The third describes the highest-level elements of the HumMod schema.

Volume 2. Structure. This volume describes the elements used in defining a model's mathematical structure. Sections include variables, equations, functions and math.

Volume 3. Control This volume describes control of the calculation of solutions using interactive control.

Volume 4. Display This volume describes the elements used in creating the display of a model's calculated solution. Tasks include selecting the variables to be displayed, selecting the display method (numerical, graph, other), and laying out the screen's panels.

Volume 5. Misc This volume might describe a variety of miscellaneous issues – but right now it focuses on file formats (schema) used for some of HumMod's XML I/O.

End

Contents

Volume 1. Model

Introducing XML

XML History

XML is an acronym for Extensible Markup Language.

This document defines the rules for using XML to document structure and details of a mathematical model in human readable and computer readable format.

Traditionally, a strange set of marks (like a wavy underline and a capital P facing backwards) were used by editors to communicate with typesetters, to ensure that a printed document was correctly laid out and attractive. This is called *markup*.

With the advent of the digital computer and electronic documents, a markup schema was needed that could be read by computers and transported across networks. In about 1969, Charles Goldfarb, Ed Mosher and Ray Lorie at IBM provided a solution in the form of GML – named after their last name initials (maybe). GML was intended to markup legal documents to send to the electronic typesetters that were rapidly replacing human typesetters. This scheme was renamed Generalized Markup Language (or so the story goes).

In the mid 1970's, Goldfarb and others added features to GML and Generalized Markup Language became Standard Generalized Markup Language (or SGML). SGML became an ISO (International Organization for Standardization) standard in 1986 and is the current standard for document markup work. But, looking ahead, it is not a markup language. It is a specification for creating markup languages and it is very complicated.

In about 1990, Tim Berners-Lee and Anders Bergland at CERN in Switzerland used SGML to define Hypertext Markup Language (or HTML) and the World Wide Web was born. HTML describes how documents should look when displayed on a computer screen. HTML was not extensible, but it was corruptible. It was a weapon used in the great browser wars (and was a casualty at that).

There remained a need to organize data as it passed not only from database to computer screen but also from database to database. And it would be nice if the scheme was much simpler than SGML. A number of people including John Bosak, Tim Bray, James Clark and C.M. Sperberg-McQueen worked in 1996 and 1997 to define a new and better markup language using SGML as the overriding standard. Extensible Markup

Language (or XML) was born. It received W3C (World Wide Web Consortium) endorsement in February 1998.

The extensible in XML's name identifies one of its major strengths. XML can be customized to meet specific needs. In the case of mathematical model documentation, we've developed an XML schema that is used to represent the details of mathematical models, including the structure, the control of solutions and the display of results.

XML Basics

XML organizes data using information stored in angle brackets (also known as the *less than* and *greater than* signs). A pair of angle brackets and the enclosed information is known as a *tag*.

A pair of tags and their enclosed data is known as an *element*. The leading tag is called the *open tag*. The enclosed data is called the *content*. The trailing tag is called the *close tag*. An example

<name> Text representing a name. </name>

The first text in an open tag is the *name* of the element. Note that no spaces are allowed between the opening angle bracket and the start of the element name.

The element name is repeated in the close tag with a forward slash (/) prefixed.

Some elements do not enclose content. All of their information is represented by a single tag. These are called *empty elements*. An example

<beep/>

An empty element is identified by a forward slash (/) as the last character before the closing angle bracket.

In addition to the element name, additional information may be stored in a tag. This information is organized as *attributes*.

Attributes are allowed in open tags and empty tags but not in close tags.

An attribute has a *name*, an equals sign and a *value*. Spaces are allowed before and after the equals sign.

Values are always enclosed in paired double (“) or single (‘) quotation marks.

Attribute values are text that can represent anything. Values typically represent names of things and numbers.

XML has some rather specific rules concerning element and attribute names. These names must begin with a letter of the alphabet or an underscore. The remaining characters in the name can be letters, digits, underscores, hyphens and periods. Names are case sensitive.

To make things easier, our schema only uses element names that are all lower case letters of the alphabet. It's also possible that no attributes will be used.

In addition to text, elements can contain other elements. An example

```
<curve>
<name> Name of curve goes here. </name>
Additional elements go here.
</curve>
```

In this example, the *curve* element contains a *name* element and some additional elements not shown.

Notice that XML uses only normal alphanumeric characters. This means that XML files can easily travel over the Internet and can also travel from one computer type to another. It also means that XML files generally cannot be infected by computer viruses.

An XML document must have a single highest-level element. It is called the *document element* or *root element*. The document element contains all of the other elements in the document (the child elements). We use *model* as the name of the document element in this schema.

An XML document may begin with a structure called an XML declaration. This looks like an XML element, but it strictly is not. It must be the first structure in the document. An example:

```
<?xml version = '1.0' ?>
```

If your XML parser requires an XML declaration, stick one in at the very top of the document.

End

Processing Instructions

Processing instructions do not describe the content of an XML document. Rather, processing instructions describe how a document's content should be processed.

Processing instructions are a private contract between the document and the parser.

The form of a processing instruction is

<?name content ?>

The *name* should follow XML's name rules. The *content* is all text after the name up to the final ?>.

HUMMOD has its own parsing algorithms. This allows us to make maximum use of processing instructions.

Several processing instructions used by HUMMOD are described next.

<?include … ?>

A document is typically assembled from more than one file. The processor must be told the file names and the sequence.

Use this instruction at the appropriate places in the document to tell the processor to include a file.

<?include Filename goes here. ?>

The DES XML parser insists that included files end at a logical break in the document. Specifically, an included file cannot end inside of a tag or in text that is being parsed for content (#PCDATA). An included file can end between elements.

<?path … ?>

When a document is assembled from files located in more than one folder, it is convenient to define a path that is prefixed to all subsequent file names.

Use this instruction to define a path to be prefixed to subsequent file names.

<?path Path specification goes here. ?>

<?create … ?>

This instruction and the next two implement conditional includes. A conditional include is an include instruction that is executed if a specific token has been created. Otherwise, the include instruction is ignored.

This multi-step process allows a single token to control multiple file includes.

Use this instruction at the appropriate place in the document (usually early on) to create a token that will control the execution of a conditional include.

<?create Token_name goes here. ?>

The token name cannot contain internal spaces. White space is used as the name delimiter.

<?if … ?>

If the named token has been created, the named file is included. Otherwise, no action is taken.

<?if Token_name File name. ?>

<?ifnot … ?>

If the named token has not been created, the named file is included. Otherwise, no action is taken.

<?ifnot Token_name File name. ?>

End

Special Characters

The *less than* (<) and *greater than* (>) characters are used to identify tags in XML. As you might imagine, if you use these characters in ordinary text, the XML parser will declare an error. Or, worse yet, it will commit an error without declaring it.

To work around this, XML defines several special characters and provides a way to represent them unambiguously and without error. Technically, the representations for special characters are called *character entities*.

Character entities begin with an ampersand (&) and end with a semicolon (;).

Ampersand	&	&amp;
Apostrophe	‘	&apos;
Greater Than	>	&gt;
Less Than	<	&lt;
Quotation Mark	“	&quot;

The table above shows the character entities for five special characters. For example, the *greater than* character can be represented in text by the character sequence *>*.

As the text is being parsed, the character entity is removed and replaced by the special character that has been specified.

The single quote (apostrophe above) and double quote (quotation mark above) are allowed in XML documents, but some restrictions can occur. An alternative representation of these two characters can come in handy.

HTML supports these character entities also and many, many more (except that the apostrophe *'* is not supported).

End

Comments

Comments can be placed in XML documents as communication to human readers. But comments might also be processed by the document parser, although this was not the original intention.

Comments may appear anywhere in a document, even before and after the document element.

The form of a comment is

```
<!-- comment -->
```

To document a file's history, put the following two comments near the beginning of the file.

```
<!-- datecreated dd-mmm-yy -->
<!-- datelastchanged dd-mmm-yy -->
```

Forbidden characters, such as angle brackets, may appear in comments. The only character sequence that is not allowed in a comment is two dashes in a row (--) that do not immediately precede the closing angle bracket. XML parsers scan for the character sequence dash-dash to locate the end of a comment, ignoring all other characters.

End

Additional Grammar

Mathematical Expressions

A mathematical expression is a set of instructions that

is used to calculate a single numerical value. This value is then used in *def*, *if* and other elements.

The goal is to have documents that can be directly parsed into an equation solver, HTML, a database or document translation tools.

Whitespace characters are space, tab and the character created by pressing the *Enter* key.

Tokens are sequences of visible characters that represent single entities within a mathematical expression. These entities can be variable names, curve names, numbers (*literal numbers*), mathematical operators, and punctuation.

All tokens in a mathematical expression must be separated by whitespace. N.B. This is an important rule. It allows us to implement a fast and efficient expression parser.

Variable And Curve Names

Rules for variable and curve names are described in the *Names* chapter.

The Order Of Calculation

Calculations inside of parentheses are completed before calculations outside of parentheses. Parentheses may be nested; calculations inside inner parentheses are completed before calculations inside outer parentheses.

Otherwise, calculations proceed from left to right.

There is no operator precedence. That is, the order of calculation is not affected by the presence or absence of a particular mathematical operator.

The expression 2 * 5 + 4 * 5 will evaluate to 70 while the expression (2 * 5) + (4 * 5) will evaluate to 30.

When in doubt, use parentheses.

Operators

Unary operators operate on a single following value. Binary operators operate on 2 values, one preceding and one following.

Mathematical operators return a decimal number. Logical operators return TRUE (1) or FALSE (0).

The names of operators are case sensitive. All operators are named using all uppercase letters.

Unary Operators

Unary mathematical operators are

-	-1.0 Multiplied By
INVERT	1.0 Divided By
EXP	Exponential Function
LOG	Natural Logarithm
LOG10	Logarithm Base 10
SQRT	Square Root
SIN	Sine

COS	Cosine
TAN	Tangent
ABS	Absolute Value
ARCSIN	Arc Sine
ARCCOS	Arc Cosine
ARCTAN	Arc Tangent
SINDEG	Sine Degrees
COSDEG	Cosine Degrees
TANDEG	Tangent Degrees
RADTODEG	Radians To Degrees
DEGTORAD	Degrees To Radians
ROUND	Round Off
ROUNDUP	Round Up
ROUNDDOWN	Round Down
FRACTPART	Fractional Part

The argument for traditional trig operators is radians.

Unary logical operators are

NOT	Logical Not

Binary Operators

Binary mathematical operators are

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Raise To The Power
MIN	Return Minimum Of
MAX	Return Maximum Of
ATAN2	Arc Tangent (X, Y) With Horz
REM	Remainder
ROUNDAT	Round At < Or > Than 1

Binary logical operators are

AND	Logical And
OR	Logical Or
EQ	Equal To
NE	Not Equal To
GT	Greater Than
GE	Greater Than Or Equal To
LT	Less Than
LE	Less Than Or Equal To

Pi

And we have one literal : PI (3.14159 …).

Functions

Functions are defined in the *functions* element. The only function just now is *curve*.

Functions are invoked by name followed by an argument name, with the name enclosed in square brackets. For example

Frank-Starling [RAP]

describes a function named *Frank-Starling* operating on a variable named *RAP*. Note the requisite spaces separating all four tokens.

Internal Variables

The equation solver will recognize and provide the correct values for the following variables. The names are treated as if they were created in the structure *System*.

System.X The model's independent variable, usually thought of as time.

System.Dx Numerical integration's integration interval. Actually used in the calculations.

System.DefaultDx Numerical integration's integration interval. Used in calculation if there is no truncation.

System.Random A random number between -1 and 1. The equation solver should generate a new random value at the beginning of each integration interval.

System.Normal A random number from a normal distribution having a mean of 0 and a standard deviation of 1. The equation solver should generate a new random, normally distributed value at the beginning of each integration interval.

System.Pi This variable holds the value of pi – 3.14 …

For more on system random numbers, see *Volume 4. Control < scramble* >.

System.Starting This variable is TRUE only once – when the equation solver is starting. Otherwise, it is FALSE.

The following intervals are for display only. Don't mess with them.

System.SolutionInterval

System.DisplayInterval

System.StorageInterval Currently not implemented.

End

Special Numbers

The equation solver can recognize the following special numbers, in addition to recognizing integer and decimal numbers and scientific notation.

TRUE This is a synonym for 1. It is the value that is returned when an expression evaluates to true. This number should be displayed as TRUE.

FALSE This is a synonym for 0. It is the value that is returned when an expression evaluates to false. This number should be displayed as FALSE.

INFINITE OK for use in logical expressions, but avoid using in mathematical expressions. This number is the result of a divide by 0. This number should be displayed as INFINITE.

UNDEFINED OK for use in logical expressions, but avoid using in mathematical expressions. This number should be displayed as UNDEFINED.

UNKNOWN OK for use in logical expressions, but avoid using in mathematical expressions. This number should be displayed as UNKNOWN.

BLANK OK for use in logical expressions, but avoid using in mathematical expressions. This number should be displayed as _____ (5 underline characters).

The advice to not use some of these numbers in mathematical expressions follows from the fact that they are very big numbers that may just trigger math processor exceptions.

Names

In this model description schema, many of the model's parts are named. This makes the specification and manipulation of names an important undertaking.

Model parts with names include variables, equations, curves, blocks of math, symbols, lists and panels.

The names involved in model structure (variables, equations, curves and blocks of math) are created within the structure element, and each structure element has a name. This fact is used to advantage.

Each name has two forms: *local* and *global*.

The local name is used to refer to a model part from within the structure where the name was created. As an example, in the structure *Exercise*, we have a variable named *ElapsedTime*.

The global name is used to refer to a model part from outside the structure where the name was created. A global name is formed by prefixing a structure name to a local name. A dot or period (.) is used as a separator between the structure and local names.

Thus exercise elapsed time is referred to as *ElapsedTime* within the structure named *Exercise* and is referred to as *Exercise.ElapsedTime* everywhere else in the XML document.

Leading and trailing spaces in names are not significant and should be stripped by the XML parser.

Interior whitespace is not allowed in names that can appear in mathematical expressions. Currently, these are the names of structures (as part of global names), variables and curves. Notice that the example developed above does not contain interior whitespace.

Some characters are not allowed in names.

- XML's special characters (see *Special Characters*).
- Any dots or periods in addition to the global name separator dot.
- Additional characters may be restricted in the future.

Top-Level Elements

Overview

An XML document has a single element (the *document element* or *root element*) that holds (or encloses) all of the document's other elements (the child elements). In this schema, the document element is named *model*.

Elements in an XML document may be nested to any level. It is customary to call elements that are the immediate children of the document element the *top-level elements*. The *model* element's top-level elements are

- schema
- modeltitle
- structure
- math
- control
- display

Structure, math, control and display are described elsewhere.

End

< schema >

Use this element to give the schema an identifier that can be used by a parser to find out which model documentation schema is being parsed.

<schema> Schema ID goes here. </schema>

This element is required. It must be the first element. Use only once per model.

An example

<schema> 2008.0 </schema>

End

< modeltitle >

Use this element to define a title for a model. I expect the equation solver will display the title on its title bar. We might also get some use out of it in Web pages and document translators.

<modeltitle> Model title goes here. </modeltitle>

This element is required. Use only once per XML document.

End

< model >

This is the model's *document element* or *root element*. It holds all of the model's other elements (the child elements).

<model> Top-level elements go here. </model>

The model element holds several types of child elements

- A schema identifier.
- A model title.
- One or more structure elements (Volume 2).
- A math element (Volume 2).
- A solution control element (Volume 3).
- Solution display elements (Volume 4).

End

Volume 2. Structure

Overview

Volume 2. Structure describes elements that are used to describe a model's structure.

The top-level element is named *structure*. It is a container element for elements that declare the variables (*variables*), declare the differential and algebraic equations (*equations*), define the functions (*functions*), and specify the math used to calculate derivatives and intermediate values (*definitions*). Each of these elements has a section in this volume.

The element *math* is also defined in this volume but it is used as a top-level element in *model*. This is not meant to be trickery. Instead, math specifications are a top-level function but block names (introduced in this volume) are an inherent part of the specifications.

End

< structure >

Use this element as a container element, holding the elements that describe the details of a model's structure. Structure detail elements are *variables*, *equations*, *functions* and *definitions*.

<structure>

<name> Unique structure name here. </name>

Structure detail elements go here.

</structure>

Variables, equations, functions, math blocks and other model components created in a structure have a *local* name (valid within the structure) and a *global* name (valid everywhere). You can review naming in *Volume 1. Model*, *Additional Grammar*, *Names*.

End

Variables

Overview

The *variables* element is used to declare and sometimes define the model's variables.

There are currently eight variable types: *var*, *parm*, *constant*, *fixedparm*, *fixedvar*, *timervar*, *normaldist*, *whitenoise*.

A variable is *declared* when it is given a name. It is *defined* when it is given a value or instructions for calculating a value. Instructions usually come in the form of a mathematical expression.

All but two of the variable types are declared and defined in a single element. The exceptions are the *var* and *fixedvar* variable types. These variables can be declared and defined in a single element, but the *var* variable type is usually declared but not initially defined. Defining takes place in a math *block* as part of the calculation of a derivative value or an intermediate value.

Variable Attributes

The table below shows the attributes of each variable type.

Defined? – Has a value. It can be referenced in a mathematical expression.

Read-Only? – Is defined and cannot be redefined.

Restarted? – Is set to its initial value when the solution is restarted.

Variable Type	Defined?	Read-Only?	Restarted?
var	Can Be	No	Yes
parm	Yes	No	Yes
constant	Yes	Yes	Yes
fixedparm	Yes	No	No
fixedvar	Usually	No	No
timervar	Yes	No	Yes
normaldist	Yes	Yes	Yes
whitenoise	Yes	Yes	Yes

System variables are defined, read-only and restarted.

< variables >

Use this element to declare and sometimes define the model's variables. There are a variety of variable types, but the two most common are an ordinary variable *var* and a parameter *parm*.

<variables> Variable declarations. </variables>

These variable types are detailed next

- var
- parm
- constant
- fixedparm
- fixedvar
- timervar
- normaldist
- whitenoise

End

< var >

Use this element to declare an ordinary variable. The variable can also be defined here (by assigning it an initial value), but defining is typically deferred to a math block.

```
<var>
    <name> Unique variable name here. </name>
    <val> Initial value (optional) here. </val>
</var>
```

Initial value is floating point.

End

< parm >

Use this element to declare and define a parameter, giving it a unique variable name and an initial value.

```
<parm>
    <name> Unique variable name here. </name>
    <val> Initial value goes here. </val>
</parm>
```

Initial value is floating point.

A parameter is a defined variable having the additional privilege of being able to receive a new value in an interactive panel via a slide bar, radio button, check box, edit box, and maybe some other visual device.

End

< constant >

Use this element to declare and define a constant, giving it a unique variable name and a value.

```
<constant>
    <name> Unique variable name here. </name>
    <val> Value goes here. </val>
</constant>
```

Value is floating point.

A constant is a defined variable having the restriction that if can't be assigned a new value. This restriction applies to <def> and <conditional> elements.

Internally, a constant has the ReadOnly variable attribute.

End

< fixedparm >

Use this element to declare and define a fixed parameter, giving it a unique variable name and an initial value.

```
<fixedparm>
    <name> Unique variable name here. </name>
    <val> Initial value goes here. </val>
</fixedparm>
```

Initial value is floating point.

A fixed parameter is a parameter having the additional attribute of not being reset to its initial conditions at solution restart. Thus, a particular value will remain in place over multiple solutions.

End

< fixedvar >

Use this element to declare and define a fixed variable, giving it a unique variable name and an (optional) initial value.

```
<fixedvar>
    <name> Unique variable name here. </name>
    <val> Initial value (optional) goes here. </val>
</fixedvar>
```

Initial value is floating point.

A fixed variable is a variable having the additional attribute of not being reset to its initial conditions at solution restart. Thus, a particular value will remain in place over multiple solutions.

The primary use of this variable type is to implement scaling in context math.

End

< timervar >

A timer variable is synchronized with the independent variable. It can count up from an initial value, count down from an initial value, or be in an OFF state. One use is to track elapsed time.

Use this element to declare and define a timer variable, giving it a unique variable name, an (optional) initial value and an (optional) initial state.

```
<timervar>
    <name> Unique variable name here. </name>
    <val> Initial value (optional) goes here. </val>
    <state> State (optional) goes here. </state>
</timervar>
```

Initial value is floating point. The default value is 0.

Valid states are UP, DOWN and OFF. The default state is OFF.

There is a *timer* element in *defininitions* that can be used to change (usually conditionally) the value and state of a timer variable.

End

< normaldist >

Use this element to declare and define a variable that gets its value from a normal distribution. The mean and standard deviation of the distribution are specified here.

```
<normaldist>
    <name> Unique variable name here. </name>
    <mean> Distribution mean here. </mean>
    <stddev> Standard deviation here. </stddev>
</normaldist>
```

Default values for mean and standard deviation are floating point 0 and 1.

There is a *scramble* element in *control* that controls randomness for this variable and the *whitenoise* variable.

End

< whitenoise >

Use this element to declare and define a variable that gets its value from a white noise distribution. The lower and upper limits of the distribution are specified here.

```
<whitenoise>
    <name> Unique variable name here. </name>
    <lowerlim> Lower limit. </lowerlim>
    <upperlim> Upper limit. </upperlim>
</whitenoise>
```

Default values for the lower and upper limits of the distribution are floating point 0 and 1.

There is a *scramble* element in *control* that controls randomness for this variable and the *normaldist* variable.

End

Equations

Overview

The *equations* element is used to declare the model's differential and implicit algebraic equations.

Equations are used to develop a relationship around several variables.

For instance, in differential equations (*diffeq*, *backwardeuler* and *stablediffeq*) the declaration names the integral and its derivative. The integral knows how to determine its own value by tracking values of its derivative.

In implicit algebraic equations (*impliciteq*) the declaration names the starting variable and ending variable. The implicit equation knows to declare a solution when the starting and ending values are nearly the same.

Automatic Declaration

When a variable is named in an equation's declaration, it is automatically declared as a *var* variable. The variable does not require additional declaration in a *variables* element.

In addition, some variables are automatically both declared and defined. I'll try to provide details in the individual equation descriptions that follow.

Calculation Sequence

Defined variables have an important role in defining other variables, since a variable cannot participate in a definition until it is itself defined.

See *Math*, *Overview* later in this volume for additional discussion of calculation sequence

End

< equations >

Use this element to declare the model's differential and implicit algebraic equations.

<equations> Equation declarations. </equations>

Declaration and partial definition takes place here. Definition is completed in *definitions* elements where derivatives and implicit equation values are calculated.

Several different equation types are currently supported.

Differential Equations

<diffeq>

<stablediffeq>

<backwardeuler>

Time Delays

<delay>

<stabledelay>

<lag>

Implicit Algebraic Equations

<impliciteq>

< diffeq >

The general form of a differential equation needing solution is

$$dY/dX = f(Y) \qquad (1)$$

where Y is the dependent variable, X is the independent variable, and f() is a functional relationship that is known. The initial value of Y is also known.

Differential equation (1) implies integral equation (2), which is the equation that is actually solved.

$$Y = \int (dY/dX)\, dX \qquad (2)$$

Use the *diffeq* element to declare and define Y and the integral in Equation (2).

```
<diffeq>
    <name> Equation name. </name>
    <integralname> Integral name. </integralname>
    <initialval> Integral's initial value. </initialval>
    <dervname> Derivative name. </dervname>
    <errorlim> Integration error limit. </errorlim>
</diffeq>
```

The equation name must be a unique name. Currently this name is used for absolutely nothing, but it may be put to use in the future.

The integral and its derivative are model variables. Thus, they must have a unique name in the variable space.

The integral's initial value and integration error limit are floating point.

The integration error limit specification determines the accuracy of the numerical integration. I typically use an integration error limit of 1% of the integral's initial value. This is arbitrary.

If the integration error limit is not specified, integration error is not controlled for this differential equation.

Two additional steps are needed to solve a differential equation

- The derivative must be defined in a math block. This is essentially an implementation of Equation (1).
- The derivative's math block must be called at the correct point in the calculation sequence.

These steps create a complete solution to Equations (1) and (2).

End

< backwardeuler >

A backward Euler integral is more stable than the standard 1st-order Euler integral.

Normally, the *diffeq* element is used to declare a differential equation. A useful alternative is available, however, that is fast and stable (but generally doesn't conserve mass). It can be used when the derivative has the form

$$dY/dt = f1 - f2 * Y \qquad (1)$$

and

$$Y = \int (dY/dX)\, dX \qquad (2)$$

Use the *backwardeuler* element to declare a differential equation. The underlying numerical method is a partially implicit 1st order Euler algorithm, commonly known as a backward Euler algorithm.

Basically, what we are doing here is replacing the derivative variable with variables representing f1 and f2 in Equation (1) above.

```
<backwardeuler>
    <name> Equation name. </name>
    <integralname> Integral name. </integralname>
    <initialval> Integral's initial value. </initialval>
```

<f1name> f1 name. </f1name>

<f2name> f2 name. </f2name>

<dervname> Derivative name. </dervname>

<errorlim> Integration error limit. </errorlim>

</backwardeuler>

Again, the equation name must be a unique name.

The integral, f1, f2 and the derivative are model variables. Thus, they all must have unique names in the variable space.

The *dervname* element is optional. It is used for display only. The f1 and f2 elements do the work.

If the integration error limit is not specified, integration error is not controlled for this differential equation.

End

< stabledelay >

A stable delay is a combination of a stable differential equation and a delay.

I won't repeat the details here. See *stablediffeq* and *delay*.

The *stabledelay* element is a *delay* element with maximum step size also specified.

```
<stabledelay>
    <name> Equation name. </name>
    <outputname> Output name. </outputname>
    <initialval> Output's initial value. </initialval>
    <inputname> Input name. </inputname>
    <rateconstname> K. </rateconstname>
    <dervname> Derivative name. </dervname>
    <errorlim> Integration error limit. </errorlim>
    <dxmaxname> Max step size. </dxmaxname>
</stabledelay>
```

Again, the equation name must be a unique name.

The output, input, rate constant, derivative and maximum step size are model variables. Thus, they all must have unique names in the variable space.

Rate constants of 0 (Tau = ∞) and INFINITY (Tau = 0) are supported.

The *dervname* element is optional. It is used for display only.

< stablediffeq >

A stable differential equation is basically a differential equation with one added feature.

The stable differential equation is used with stiff systems.

If we have physical or observational evidence that an equation will become unstable at larger step sizes, we pass this information to the stable differential equation; it will then attempt to take care of stability.

```
<stablediffeq>
    <name> Equation name. </name>
    <integralname> Integral name. </integralname>
    <initialval> Integral's initial value. </initialval>
    <dervname> Derivative name. </dervname>
    <errorlim> Integration error limit. </errorlim>
    <dxmaxname> Max step size. </dxmaxname>
</stablediffeq>
```

The equation name must be a unique name. Currently this name is used for absolutely nothing, but it may be put to use in the future.

The integral, its derivative and maximum step size are model variables. Thus, they must have a unique name in the variable space.

The integral's initial value and integration error limit are floating point.

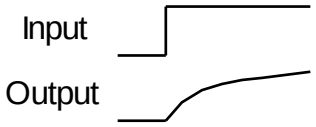
The integration error limit specification determines the accuracy of the numerical integration. I typically use an integration error limit of 1% of the integral's initial value. This is arbitrary.

If the integration error limit is not specified, integration error is not controlled for this differential equation.

End

< delay >

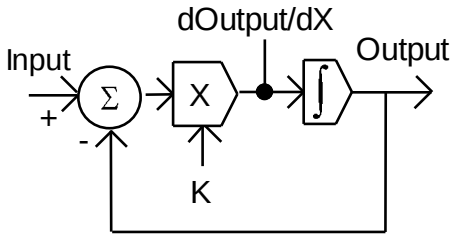
The output of a delay slowly follows its input as illustrated in the figure below.



The most common delay is called 1st-order. It is implemented with a differential equation as shown in Equations (1) and (2) and the block diagram below.

$$\text{Output} = \int (\text{dOutput/dX}) * \text{dX} \qquad (1)$$

$$\text{dOutput/dX} = K * (\text{Input} - \text{Output}) \qquad (2)$$



The rapidity of the output's response is determined by the rate constant, which is denoted above by K.

If time is the independent variable, a delay is called a *time delay*.

Use the *delay* element to create a delay.

<delay>

<name> Equation name. </name>

<outputname> Output name. </outputname>

<initialval> Output's initial value. </initialval>

<inputname> Input name. </inputname>

<rateconstname> K. </rateconstname>

<dervname> Derivative name. </dervname>

<errorlim> Integration error limit. </errorlim>

</delay>

Again, the equation name must be a unique name.

The output, input, rate constant and derivative are model variables. Thus, they all must have unique names in the variable space.

Rate constants of 0 (Tau = ∞) and INFINITY (Tau = 0) are supported.

The *dervname* element is optional. It is used for display only.

If the integration error limit is not specified, integration error is not controlled for this differential equation.

End

< lag >

Use this element to create a lag. Like a delay, the output variable follows the input variable by a 1st-order delay that is determined by the rate constant.

But there is a difference here. A lag output variable must be referred to in a math block to define it. The reference must be at a place where the input variable is defined and the output variable is needed. The *calclag* element in *definitions* is used to create the proper reference.

```
<lag>
    <name> Equation name. </name>
    <outputname> Output name. </outputname>
    <initialval> Output's initial value. </initialval>
    <inputname> Input name. </inputname>
    <rateconstname> K. </rateconstname>
    <dervname> Derivative name. </dervname>
</lag>
```

Again, the equation name must be a unique name.

The output, input, rate constant and derivative are model variables. Thus, they all must have unique names in the variable space.

Rate constants of 0 (Tau = ∞) and INFINITY (Tau = 0) are supported.

The *dervname* element is optional. It is used for display only.

There is no integration error limit.

This is a rather weird numerical method, but it can be very useful in introducing a little delay into a response. I think that it can be shown that a lag becomes more stable as step size increases – but this is a topic for elsewhere.

End

< impliciteq >

An implicit algebraic equation has the form of

$$Y = f(Y) \qquad (1)$$

This equation is solved using two estimates of Y: the beginning Y (referred to as YStart here) and the ending Y (referred to as YEnd). Equation (1) becomes

$$YEnd = f(YStart) \qquad (2)$$

A solution has been obtained when

$$| YEnd - YStart | <= Error\ Limit \qquad (3)$$

where the vertical bars denote absolute value and the error limit is the largest difference that is acceptable.

Use this element to declare an implicit algebraic relationship. The lead variable is YStart in Equation (2).

<impliciteq>

<name> Equation name. </name>

<startname> Start name. </startname>

<initialval> Start's initial value. </initialval>

<endname> End name. </endname>

<errorlim> Solution error limit. </errorlim>

<searchminname> Min name. </searchminname>

<searchmin> Min value. </searchmin>

<searchmaxname> Max name. </searchmaxname>

<searchmax> Max value. </searchmax>

</impliciteq>

We can improve search efficiency by constraining the search. Constraint also allows us to get the correct root when more that one root exists. If the minimum and/or maximum limits are not specified, the search is not constrained in the respective direction.

We can limit the minimum value of the search space in any of three ways.

- Name a variable whose value limits the search space.
- Specify a floating point value that limits the search space.
- Specify nothing and get no constraint.

The maximum value of the search space can be limited in the same way.

Two additional steps are needed to fully solve an implicit algebraic equation.

- The relationship between the start variable (YStart) and end variable (YEnd) must be fleshed out in an *implicit* math block. The start variable is assumed to be defined at the start of the block and the end variable must be defined by the end of the block. This is an implementation of Equation (2) above.

- The implicit math block must be called at the correct point in the calculation sequence.

Functions

Overview

The *functions* element is used to define the model's functions. Currently the only function that is supported is the *curve*.

A *curve* is a function that takes a single input and returns a single output. The curve is used in mathematical expressions. The curve's functional relationship is represented by modified cubic splines.

There is an alternative way to implement the functionality of a function. The alternative is to use a sequence of *copy*, *call* and *copy* elements. This approach is useful, powerful and verbose. I'll try to provide details in *definitions*.

End

< functions >

Use this element to define a model's functions. The one available function type is *curve*.

<functions> Function definitions here. </functions>

The functions element is not needed in structures with no function definitions.

End

< curve >

A curve is a function that takes a single input and returns a single output in a mathematical expression.

The curve element contains child elements that name the curve and list the data points that define the curve.

Each data point contains an x value, y value and slope.

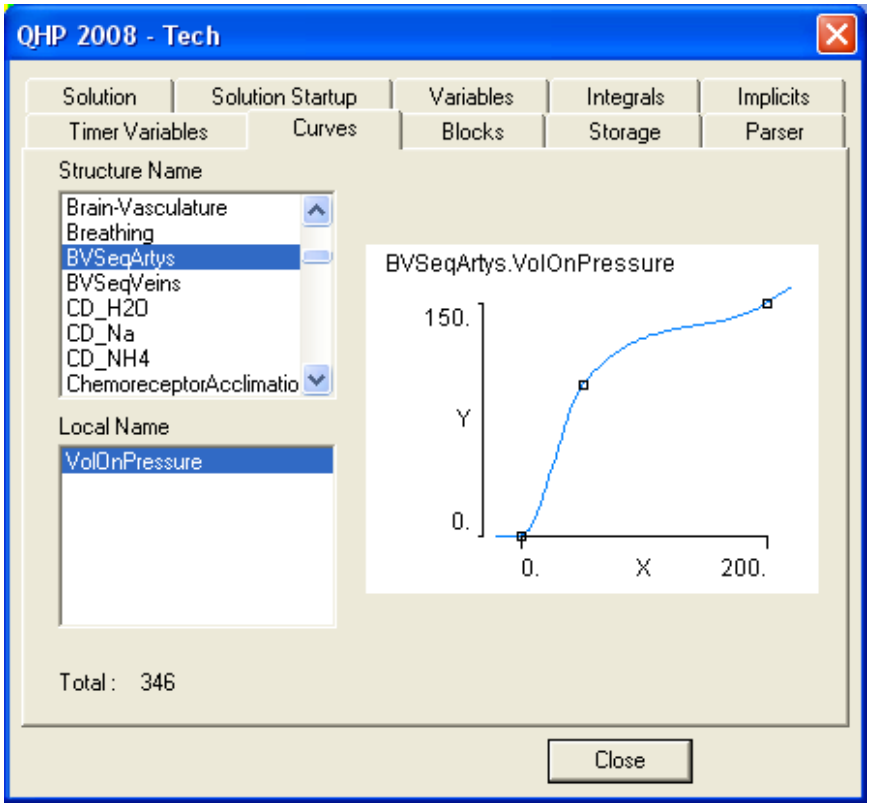
```
<curve>
    <name> Curve name. </name>

    <point>
        <x> X </x> <y> Y </y> <slope> S </slope>
    </point>

    More points go here.

</curve>
```

X, Y and S are floating point.

- A valid curve definition has two or more data points. There is no upper limit to the number of data points.
- Curve X values must be defined in ascending order.

As an example, here is a curve named VolOnPressure taken from HUMMOD 2008, BVSeqArtys.DES.

The curve is shown below as it appeared in the Tech tabbed dialog box.



The curve definition is

```
<curve>
    <name> VolOnPressure </name>
    <point>
        <x>  0 </x><y>  0 </y><slope> 0 </slope>
    </point>
```

```
        <point>
            <x>  50 </x><y>  97 </y><slope> 1.0 </slope>
        </point>
        <point>
            <x> 200 </x><y> 150 </y><slope> 0.5 </slope>
        </point>

    </curve>
```

The curve was used in the following *def* element.

```
    <def>
        <name> TMP </name>
        <val> VolOnPressure [ Vol ] </val>
    </def>
```

In the *val* element, the name of the curve is followed by square brackets enclosing the name of the input variable. The output value is TMP (for transmural pressure). The input value is Vol (for blood volume).

End

Definitions

Overview

The *definitions* element specifies the math used to calculate the model's derivatives and intermediate variables.

The math specifications are grouped into named *block*s. The information in the blocks describes the calculations and much of the calculation sequence.

The *math* element will be presented in another section. It is used to describe how the calculations are started.

A couple of the workhorse elements here are *def* which defines a variable and *call* which invokes a block of definitions.

End

< definitions >

Use this element as a container for the blocks of math used to calculate the model's derivatives and intermediate variables.

<definitions> Blocks of math. </definitions>

End

< block >

Use this element to define a named block of math.

```
<block>
    <name> Unique block name. </name>
    Many different math elements can go here.
</block>
```

A block of math is invoked (or called) using the *call* element.

End

< call >

Use the *call* element to call or invoke a *block* of math in *definitions*.

<call> Block name goes here. </call>

End

< implicitmath >

Use this element to define the math that is used to solve an implicit algebraic equation.

Here is a brief review, taken from the *Equations* section of this volume.

An implicit algebraic equation has the form of

$$Y = f(Y) \qquad (1)$$

This equation is solved using two estimates of Y: the beginning Y (referred to as YStart here) and the ending Y (referred to as YEnd). Equation (1) becomes

$$YEnd = f(YStart) \qquad (2)$$

A solution has been obtained when

$$| YEnd - YStart | <= Error\ Limit \qquad (3)$$

where the vertical bars denote absolute value and the error limit is the largest difference that is acceptable.

When an *impliciteq* equation element is created, variables representing YStart and YEnd above are named (declared). The next step, implementing Equation (2), takes place here.

Use the *implicitmath* element to create the math that fleshes out Equation (2). The starting variable, named in the *impliciteq* declaration, is defined at the start of the element. The ending variable, also named in the *impliciteq* declaration, must be defined by this math by the end of the element.

Equation (3) above is automatically evaluated at the end of the element. The element is iterated until an acceptably small error is obtained.

```
<implicitmath>
    <name> Implicit equation name. </name>

    The YStart variable is defined here. Now do the math needed to
    define YEnd.

</implicitmath>
```

End

< andif >

I took the following paragraph from the *if* element documentation.

Note that our parser does not support recursion, so nested *if* elements are not allowed in this schema. If you need an *if* element within an *if* element, use a *andif* element for the 2nd if.

So here is the *andif* element. Use this element to implement conditional math in situation where the *if* element is not allowed.

The mathematical expression in *test* is evaluated. If it evaluates to TRUE, the math in the *true* element is executed. If it evaluates to FALSE, the math in the *false* element is executed.

```
<andif>
    <test> Evaluate this expression. </test>
    <true> Math for TRUE outcome. </true>
    <false> Math for FALSE outcome. </false>
</andif>
```

The true or false element is not needed if there is no conditional math for it to do.

End

< beep >

When this element is encountered during the calculation of a solution, it signals the equation solver that an audio beep is needed.

A beep element is generally only useful in conditional math or when trying to straighten out tangled logic.

```
<beep/>
```

End

< calclag >

A *lag* equation is like a *delay* equation. The output variable follows the input variable by a 1st-order delay that is determined by the rate constant.

But there are differences. A *delay* equation creates a delay in a model. A *lag* equation creates a delay in an algebraic relationship. This means that a *delay* equation acts like a modified differential equation while a *lag* equation acts like a modified algebraic variable.

The *lag* equation's lead variable is the output or delayed variable. It must be defined in a *definitions* element just like any other variable. Think of this as

Lag Output = k * Lag Input

where k is a magic coefficient that produces a delay.

In *definitions*, at any point after the lag's input variable has been defined, define the lag's output variable by referring to the *lag* using the *calclag* element and the lag's name.

<calclag> Lag name goes here. </calclag>

You'll get a run time error if the *calclag* element is invoked before its input variable is defined.

End

< conditional >

Use the *conditional* element to implement conditional math.

The mathematical expression in *test* is evaluated. If it evaluates to TRUE, the expression in *true* is evaluated and the result is assigned to the named variable. If it evaluates to FALSE, the expression in *false* is evaluated and the result is assigned to the named variable.

```
<conditional>
    <name> Variable name. </name>
    <test> Evaluate this expression. </test>
    <true> Exression for TRUE outcome. </true>
    <false> Exression for FALSE outcome. </false>
</conditional>
```

The *true* or *false* elements are not needed if there is no assignment to make.

End

< copy >

We can use a math block as a multiple input – multiple output funtion. Use the *copy* element to copy values into the block, *call* the block, and use the *copy* element a second time to copy values back out of the block.

```
<copy>
    <from> Source variable name. </from>
    <to> Destination variable name. </to>
</copy>
```

End

< def >

Use the *def* element to define a declared variable that has not yet been defined. Define means to assign a value or instructions for calculating a value (a mathematical expression).

```
<def>
    <name> Variable name. </name>
    <val> Mathematical expression. </val>
</def>
```

Evaluate the mathematical expression and assign the result to the named variable.

If this was a traditional assignment statement, it would look like this

    Variable name = Mathematical expression

End

< exitblock >

When this element is encountered during the calculation of a solution, it triggers an immediate exit from the block that is being calculated.

An *exitblock* element can be useful in conditional math that has a rather complex organization.

```
<exitblock/>
```

End

< if >

Use this element to implement conditional math.

The mathematical expression in *test* is evaluated. If it evaluates to TRUE, the math in the *true* element is executed. If it evaluates to FALSE, the math in the *false* element is executed.

```
<if>
    <test> Evaluate this expression. </test>
    <true> Math for TRUE outcome. </true>
    <false> Math for FALSE outcome. </false>
</if>
```

The true or false element is not needed if there is no conditional math for it to do.

Note that our parser does not support recursion, so nested *if* elements are not allowed in this schema. If you need an *if* element within an *if* element, use a *andif* element for the 2nd if.

End

< logevent >

This element asks the equation solver to update storage and refresh the display between regularly scheduled display updates.

Some values are of interest, but are not displayed because they exist within a display interval rather than at the end of the interval. This element can be used to conditionally capture a value that would otherwise be missed, such as a maximum or minimum when a variable suddenly changes direction.

The independent variable will be advanced to exactly the value specified by the named variable and storage and display will be updated at that time.

If the named variable is the independent variable (*System.X*) then the logging takes place immediately.

The *logevent* element is generally only useful in conditional math.

<logevent>

Name of variable specifying logging time (or technically specifying logging value of independent variable) goes here.

</logevent>

✓ Valid in *%math*.

✓ DTD

<!ELEMENT logevent (#PCDATA)>

End

< message >

Use this element to define a message to be displayed by the equation solver. A *message* is meant to be displayed immediately while a *page* is meant to be displayed after the completion of a successful integration interval.

A message is generally only useful in conditional math.

<message> Message text here. </message>

The message element is sometimes useful to the model builder when the model's logic is not clear (i.e. use it for debugging).

End

< onjustchanged >

Sometimes a parameter feeds some dependent variables that should be recalculated when the parameter is assigned a new value. But the dependent variables should not be recalculated otherwise. Use this element to implement such a scenario.

<onjustchanged>

<name> Parameter name goes here. </name>

Math goes here. Math will be executed if the named parameter has just been assigned a new value.

</onjustchanged>

End

< ontimedout >

Sometimes we set a timer variable and then want to take some action when the timer variable times out (i.e. counts up or down to zero). The action is typically turning on or off some regimen.

Use the *ontimedout* element to implement such a scheme.

<ontimedout>

<name> Timer variable name. </name>

Math goes here. Math will be executed if the named timer variable has just timed out (counted up or down to zero).

</ontimedout>

Note that timer variables can only time out in wrapup code – where time is known with certainly.

End

< page >

Use this element to define a message to be displayed by the equation solver. A *page* is meant to be displayed after the completion of a successful integration interval while a *message* is meant to be displayed immediately.

A *page* is generally only useful in conditional math.

```
<page> Pager message. </page>
```

Pages are queued, so multiple pages can appear at the end of a successful integration interval.

HUMMOD uses pages extensively to report unexpected events.

End

< stop >

When this element is encountered during the calculation of a solution, it signals the equation solver that the solution should be stopped.

A stop element is usually only useful in conditional math.

```
<stop/>
```

End

< testcase >

Use the *testcase* element as a container for an arbitrary number of *case* elements.

The *test* expression in each case is evaluated until a TRUE value is obtained. Then the math elements following the *test* element are executed and *testcase* is exited.

Note that this logic means that only one or none *case*s will be executed per iteration.

```
<testcase>

<case>
    <test> Evaluate this expression. </test>

    If the expression evaluates to TRUE, do this math.

</case>

More cases go here.

</testcase>
```

Sometimes we want the final case to be executed if none of the other cases are executed (i.e. execute the default case). To implement this, simply make the test expression TRUE.

```
<test> TRUE </test>
```

End

< timer >

Use this element to change the value and state of a *timervar*. This element is designed for use in conditional math, supplying the appropriate response when a pump, treadmill or something similar is turned on or off.

```
<timer>
    <name> Timer variable name. </name>
    <val> New value via expression. </val>
    <state> New state. </state>
</timer>
```

Valid states are UP, DOWN and OFF.

If the value or state is not being changed, the appropriate element can be omitted.

End

Math

Overview

The *definitions* element is a container element that holds blocks of instructions. These blocks define the calculations and much of the calculation sequence.

The *math* element is then used to specify how the calculations should be started.

The equation solver does four types of math.

- Context Math This math is calculated just once at the beginning of a solution. The math is primarily used to scale the model.

- Parameter Math This math is calculated after a parameter value has been changed. This math defines values that a solely derivatives of parameter values.

- Derivative Math This math is used to calculate the derivatives during each integration interval. This is the workhorse.

- Wrapup Math This math is calculated after the successful completion of an integration interval. Variables not used in calculating the derivatives can be defined here, such as variables used only in display. Variables that might cause a discontinuity in the solution are also defined here.

The equation solver then wants to know the names of the four blocks that initiate these four types of calculations. We use the math element to specify the four appropriate block names.

Having a block of each calculation type is optional. Of course, if we have no named blocks, no calculations will be undertaken.

The order of frequency of calculation types (most to least) is

- Derivatives
- Parameters
- Wrapup
- Context

Calculation Sequence

The sequence of calculation is significant. The technical rule is that each variable must be *defined* before it can be *referred to*. This keeps numerical errors small.

Referred to means being named in an equation on the right-hand side of the equals sign.

Defined means being named in an equation on the left-hand side of the equals sign.

In this XML schema, the *def* element is equivalent to an equation.

```
<def>
    <name> Variable Name </name>
    <val> Mathematical Expression </val>
</def>
```

The *name* variable name is on the left-hand side of the equals sign and is defined in the *def* element. The *val* mathematical expression is on the right-hand side of the equals sign. It can refer only to variables that are defined.

As an example, consider the equations

A = 20 * B	(1)
B = 5 * C	(2)

These two equations cannot be calculated in the order shown above, because *B* is referred to in Equation (1) before it is defined in Equation (2). So the order of calculation must be reversed, with (2) being calculated before (1). In addition, *C* must be defined earlier yet to make it available when (2) is calculated.

Use the order of *def*'s within a block and the *math* and *call* elements across blocks to specify the correct order of calculation.

End

< math >

Use this element to specify how the calculation sequence starts off in several different calculation categories. A math block is specified for each calculation category.

There are four types of calculations: *context*, *parms*, *dervs* and *wrapup*.

```
<math>
    <context> Context math block. </context>
    <parms> Parameter math block. </parms>
    <dervs> Derivative math block. </dervs>
    <wrapup> Wrapup math block. </wrapup>
</math>
```

The math element can be omitted when there are no relevant calculations to call (which is never).

The various math elements are optional. There are no defaults.

I've snipped the *math* element from HUMMOD 2008, Model.DES.

```
<math>
    <context> Context.Math </context>
    <parms> Structure.Parms </parms>
```

```
        <dervs> Structure.Dervs </dervs>
        <wrapup> Structure.Wrapup </wrapup>
    </math>

End
```

Volume 3. Control

Overview

Volume 3. Control describes elements that are used to control a solution.

Several forms of control will eventually be implemented, including scripted control and remote control. But in this part of the schema, we are referring only to interactive control – which is the normal way that users control HUMMOD 2008 solutions.

The top-level control element is named *control*. It is a container element for child elements that, for the most part, populate the Go pop-down menu.

Intervals

A *solution* is all of the calculated values that accumulate as the model's independent variable is advanced.

A solution is defined by five interrelated intervals.

The *cumulative interval* is the advance in the independent variable (usually time) from start to finish. More specifically, the cumulative interval is the sum of all solution intervals. Cumulative intervals are generally determined by the user's interactive choices.

The *solution interval* is the advance in the independent variable from a pause in the calculation to the next pause in the calculation. Solution intervals are selected by the user from choices presented on the *Go* menu. As noted above, solution intervals are strung together end to end to create the cumulative interval. It follows that the solution interval is less than or equal to the cumulative interval.

The *display interval* is the advance in the independent variable between display updates. The display interval determines the frequency of screen refreshes. The display interval is less than or equal to the solution interval.

The *storage interval* is the advance in the independent variable between storage updates. The storage interval determines the granularity of the displayed solution. The storage interval is less than or equal to the display interval. It the current implementation we don't use a storage interval; instead, we store all solution values.

The *integration interval* is the advance in the independent variable used in numerical integration. The integration interval is determined by the equation solver and is less than or equal to the storage interval. The size of the integration interval is determined by the equation solver's estimate of the numerical integration error.

End

< control >

Use this element as a container element for solution control elements. The control element is valid only as a child of the *model* element.

<control>

Child control elements go here.

The main control elements are *gofor*, *goto* and *gobar*.

Miscellaneous control elements are *initialx*, *scramble* and *settlingtime*.

</control>

End

< gobar >

Use this element to add a separator bar to the Go popdown menu, as illustrated at right.

<gobar/>

The separator bars in Vista are very faint – acting almost as a space. The visual at right is from an older version of Windows.

A separator bar (using a different element name) can also be added to panel pop-down menus.

End

< gofor >

Use this element to create a Go menu item that can be used to advance the solution. A Go menu is shown at right.

Specify the solution and display intervals and define a menu item for the Go popdown menu.

Intervals are specified in floating point.

Some typical menu items are shown at right.

There are default values. The default *displayint* is *solutionint* / 10. The default *menuitem* is an image of the *solutionint*.

<gofor>

<solutionint> Solution interval. </solutionint>

<displayint> Display interval. </displayint>

<menuitem> Go menu item. </menuitem>

</gofor>

We have also had a storage interval. But right now all of the numerical values in a solution are stored, so the storage interval specification is not implemented.

End

< goto >

The *goto* element is very similar to the *gofor* element, with one exception as explained below.

Use this element to create a Go menu item that can be used to advance the solution. A Go menu is shown at right. The bottom four menu items were generated using the *goto* element.

Specify the solution and display intervals and define a menu item for the Go popdown menu.

Intervals are specified in floating point.

There are default values. The default *displayint* is *solutionint* / 10. The default *menuitem* is an image of the *solutionint*.

This is a little tricky, but it works great in tidying up the progress of a solution. A solution interval is specified, but the solution doesn't necessarily go the full interval. Instead, it goes until the independent variable (*System.X*) has increased to the next multiple of the specified solution interval.

Thus, this is a modified *gofor* element.

<goto>

<solutionint> Solution interval. </solutionint>

<displayint> Display interval. </displayint>

<menuitem> Go menu item. </menuitem>

</goto>

As an example, suppose the independent variable has a current value of 2000. The Go menu has a go to menu item with a solution interval of 1440 (and maybe a label of *To Next Day*). When user clicks this choice, the solution will be advanced to the next multiple of 1440, which is 2880, for a net advance of 880.

End

< initialx >

Use this element to define the initial value of the equation solver's independent variable, called System.X, usually representing time. The default, and most common, value is 0.

<initialx> Initial value of X goes here. </initialx>

The value is floating point.

End

< scramble >

The random number generator is always seeded the same way. Thus, it generates a random, but predictable, sequence during repeat solutions.

Use this (empty) element to switch to a truly random seed. The random number generator will generate a new random sequence for each solution.

<scramble/>

End

< settlingtime >

There are two ways to start HUMMOD.

- Start with the specified initial conditions. This occurs when settling time is not specified.

- Start with the specified initial conditions and advance the solution a little bit to get a better estimate of steady-state. The values at the end of this settling time are subsequently (and automatically) used as the initial conditions. The settling time in minutes (floating point) is specified using the following element.

<settlingtime> Settling time here. </settlingtime>

I've been using 200 minutes as a settling time, but this is completely arbitrary.

End

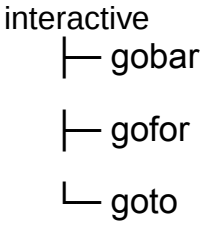
Interactive

Overview

This section defines elements that are used to control the solution through the user's interactive choices.

These elements help to populate a popdown menu (the *Go* menu) that gives the user a choice of solution intervals.

The highest-level interactive control element is named *interactive*. It is a container element for the interactive control children.



The processing instructions *include*, *prefixpath*, and *strict* may also be used in *interactive* where appropriate. See *Volume 1. Model*, *Processing Instructions*.

End

< gobar >

Use this element to add a separator bar to the Go popdown menu, shown at right.

<gobar/>

✓ Valid in < interactive >

✓ See also

< gofor >
< goto >

✓ DTD

<!ELEMENT gobar EMPTY>

End

< goto >

See this volume's *Overview* for a description of intervals.

Use this element to create a Go menu item that can be used to advance the solution. A Go menu is shown at right. Note the bottom four menu items.

Specify the solution, display and storage intervals and define a menu item for the Go popdown menu.

This is a little tricky, but it works great in tidying up the progress of a solution. A solution interval is specified, but the solution doesn't go the full interval. Instead, it goes until the independent variable (*System.X*) has increased to the next multiple of the specified solution interval.

Thus, this is a modified *gofor* element.

<goto>

<solutionint>

Solution interval goes here.

</solutionint>

<displayint>

Display interval goes here.

</displayint>

<storageint>

Storaage interval goes here.

</storageint>

<menuitem>

Go menu item goes here.

</menuitem>

</gofor>

As an example, suppose the independent variable has a current value of 2000. The Go menu has a goto menu item with a solution interval of 1440 (and maybe a label of *To Next Day*). When user clicks this choice, the solution will be advanced to the next multiple of 1440, which is 2880, for a net advance of 880.

✓ Valid in < interactive >

✓ See also

< gofor >

< gobar >

✓ DTD

<!ELEMENT gofor

(solutionint,
displayint?,
storageint?,
menuitem?
)
>

<!ELEMENT solutionint (#PCDATA)>

<!ELEMENT displayint (#PCDATA)>

<!ELEMENT storageint (#PCDATA)>

<!ELEMENT menuitem (#PCDATA)>

The default *displayint* is *solutionint* / 10. The default *storageint* is *displayint*. The default *menuitem* is an image of the *solutionint*.

End

< gofor >

See this volume's *Overview* for a description of intervals.

Use this element to create a Go menu item that can be used to advance the solution. A Go menu is shown at right.

Specify the solution, display and storage intervals and define a menu item for the Go popdown menu.

<gofor>

<solutionint>

Solution interval goes here.

</solutionint>

<displayint>

Display interval goes here.

</displayint>

<storageint>

Storaage interval goes here.

</storageint>

<menuitem>

Go menu item goes here.

</menuitem>

</gofor>

✓ Valid in < interactive >

✓ See also

< goto >
< gobar >

✓ DTD

<!ELEMENT gofor

(solutionint,

displayint?,

storageint?,

menuitem?

)

>

<!ELEMENT solutionint (#PCDATA)>

<!ELEMENT displayint (#PCDATA)>

<!ELEMENT storageint (#PCDATA)>

<!ELEMENT menuitem (#PCDATA)>

The default *displayint* is *solutionint* / 10. The default *storageint* is *displayint*. The default *menuitem* is an image of the *solutionint*.

End

< interactive >

Use this element as a container element for the interactive control elements.

<interactive>

Interactive control elements go here.

</interactive>

✓ Valid in < control >

✓ See also < scripted >

< remote >
< misc >

✓ DTD The elements in *interactive* may be defined in any amount but order is significant. Order determines the order of menu items in the *Go* popdown menu.

<!ELEMENT interactive

(gobar

| gofor

| goto

)*

>

Misc

Overview

This section describes elements that are not part of interactive control or scripted control. Thus, they are miscellaneous.

misc
- initialx
- scramble

Elements *include*, *prefixpath*, *datecreated* and *datelastaltered* may also be used in *structure* when appropriate. See *Volume 1. Model*, *Top-Level Elements*.

End

< misc >

Use this element as a container element for miscellaneous control elements.

<misc>

Miscellaneous control elements go here.

</misc>

✓ Valid in < control >

✓ See also < interactive>

< scripted >

✓ DTD Some elements in *misc* may be defined only once.

<!ELEMENT misc

((#PCDATA,
initialx?,
#PCDATA,
scramble?,
#PCDATA
)

```
        |
        (   #PCDATA,
            scramble?,
            #PCDATA,
            initialx?,
            #PCDATA
        ))
    >

    End
```

< initialx >

Use this element to define the initial value of the equation solver's independent variable, called System.X, usually representing time. The default, and most common, value is 0.

```
<initialx>
    Initial value of X goes here.
</initialx>
```

✓ Valid in < misc >

✓ DTD

```
<!ELEMENT initialx (#PCDATA)>
```

End

< scramble >

The random number generator is always seeded the same way. Thus, it generates a random, but predictable, sequence during repeat solutions.

Use this element to switch to a random seed. The random number generator will generate a truly random sequence during each solution.

<scramble/>

✓ Valid in < misc >

✓ See also

< whitenoise >
< normaldist >

in *Volume 2. Structure … Variables*

✓ DTD

<!ELEMENT scramble EMPTY>

End

Remote

Overview

This section defines the elements that implement remote requests for processing submitted scripts.

The highest-level remote control element is named *remote*. It is a container element for the remote control children.

remote
- openlistener
- closelistener
- changewindow
- logfile
- closesolver

A *listener* periodically looks for the existence of a specific file. When the listener finds the file, it processes the file.

A remote user may establish its own listener, with appropriate filename and listening interval.

To start (bootstrap) the process, we must first open basic listener that listens for a standard filename.

I arbitrarily recommend a basic listener named *BasicListener* that listens for a file name *BasicListener.DAT* at an interval of 5 seconds.

A listener does not parse the entire *model* XML schema. Instead, it parses three out the four child elements in *control*.

A new *document element*, named *remoterequest*, is used in this remote request schema.

remoterequest
- scripted
- remote

└ misc

A remote request, then, is primarily a script.

End

< remote >

Use this element as a container element for all of the remote control elements.

<remote>

Remote control elements go here.

</remote>

✓ Valid in < control >

✓ See also < interactive >

< scripted >

< misc >

✓ DTD The elements in *remote* may be defined in any order and amount.

<!ELEMENT remote

(openlistener

| closelistener

| changewindow

| logfile

```
            | closesolver
        )*
    >

    End
```

< openlistener >

Use this element to open a listener.

A listener periodically (once each interval) checks for the existence of a specific (target) file. When the listener finds the file, it processes the file.

```
<openlistener>
    <name>
        Listener's name goes here.
    </name>
    <filename>
        Target filename goes here >.
    </filename>
    <interval>
        Listening interval (seconds) goes here.
    </interval>
</openlistener>
```

We refer to the listener's name when closing the listener.

✓ Valid in *remote*.

✓ DTD

```
<!ELEMENT openlistener (name, filename, interval)>
```

<!ELEMENT name (#PCDATA)>

<!ELEMENT filename (#PCDATA)>

<!ELEMENT interval (#PCDATA)>

End

< closelistener >

Use this element to close down a listener.

Listeners consume resources. If you are done using a listener, close it down.

```
<closelistener>
    Listener name goes here.
</closelistener>
```

✓ Valid in *remote*.

✓ DTD

```
<!ELEMENT closelistener (#PCDATA)>
```

End

< logfile >

Use this element request a log file. A log file is used to log the progress made when a script is processed.

The file is created when the processing is completed.

The presence of this file tells a remote user that the script has been processed and the script output file is now ready for parsing.

The content of this file indicates success or failed and it shows the errors, if any, that were logged.

See *Volume 5. Misc*, *File Formats* for the file format.

```
<logfile>
    Log file name goes here.
</logfile>
```

✓ Valid in *remote*.

✓ DTD

```
<!ELEMENT logfile (#PCDATA)>
```

End

< closesolver >

Use this element to close down the equation solver.

If a remote user launched the equation solver and is now done using it, this is the proper way to shut it down.

```
<closesolver>
```

✓ Valid in *remote*.

✓ DTD

```
<!ELEMENT closesolver EMPTY>
```

End

< changewindow >

Use this element to change the state of the equation solver's main window.

```
<changewindow>
    Window action goes here.
</changewindow>
```

Possible actions are HIDE, SHOW, MINIMIZE and RESTORE.

✓ Valid in *remote*.

✓ DTD

```
<!ELEMENT changewindow (#PCDATA)>
```

End

Scripted

Overview

This section defines elements that can be part of a script that controls the solution.

Multiple solutions may be calculated and selected data can be saved to data files. User interaction is not necessarily required.

The highest-level scripted control element is named *scripted*. It is a container element for the scripted control children.

scripted
- fixedparm
- %tasks

%tasks
- accept
- advancefor
- advanceto
- call
- def
- eval
- exitscript
- fileaccept
- fileannotate
- fileclose
- filenewline
- fileopenappend
- fileopencreate

├─ fileroster

├─ filestarttracking

├─ filestoptracking

├─ filetimestamp

├─ fileupdate

├─ fileverify

├─ filewriteheader

├─ loadics

├─ message

├─ pause

├─ repeatfor

├─ repeatwhile

├─ replacenote

├─ reset

├─ restart

├─ saveics

├─ selectpanel

├─ setdisplayspeed

├─ setfileattribute

├─ setpagerstatus

└─ setworkdir

End

< fixedparm >

Use this element to declare and define a variable to be used in a script.

This variable type was chosen because it does not lose its value on a solution restart. Thus, it can be used in the control of multiple solutions. See also *Volume 2. Structure*, *Variables* where *fixedparm* is also an element.

```
<fixedparm>
    <name>
        Unique variable name goes here.
    </name>
    <val>
        Parameter's initial value goes here.
    </val>
</fixedparm>
```

✓ Valid in < scripted >.

✓ DTD

<!ELEMENT fixedparm (name, val)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT val (#PCDATA)>

End

%tasks

Most of the scripted elements appear as children in several different scripted elements. These have been grouped in the *%tasks* entity.

See *Volume 5. Docs*, *Document Type Description* for a description of entities.

✓ DTD This entity is designed to be content in an element declaration. It states that a variety of child scripted elements may appear in an element in any order and amount.

<!ENTITY % tasks

"(accept

| advancefor

| advanceto

| call

| def

| eval

| exitscript

| fileaccept

| fileannotate

| fileclose

| filenewline

| fileopenappend

| fileopencreate

```
            | fileroster
            | filestarttracking
            | filestopstracking
            | filetimestamp
            | fileupdate
            | fileverify
            | filewriteheader
            | loadics
            | message
            | pause
            | repeatfor
            | repeatwhile
            | replacenote
            | reset
            | restart
            | saveics
            | selectpanel
            | setdisplayspeed
            | setfileattribute
            | setpagerstatus
            | setworkdir
        )*"
    >

    End
```

< accept >

Use this element to check the value of the named variable. If the value is less than the minimum specified or greater that the maximum specified, a not accepted message is displayed.

<accept>
<name>
Name of variable to be evaluated goes here.
</name>
<min>
Minimum acceptable value goes here.
</min>
<max>
Maximum acceptable value goes here.
</max>
By default, the script continues after announcing an unacceptable value. But, you can use the empty element below to stop the script after an unacceptable value has been announced.
<stoponnotaccepted/>
</accept>

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT accept

(name,

```
        min?,
        max?,
        stoponnotaccepted?
    )
>

<!ELEMENT name (#PCDATA)>
<!ELEMENT min (#PCDATA)>
<!ELEMENT max (#PCDATA)>
<!ELEMENT stoponnotaccepted EMPTY>
```

If *min* is not specified, the lower bounds are not checked. If *max* is not specified, the upper bounds are not checked. If *stoponnotaccepted* is not specified, the script continues.

End

< advancefor >

See this volume's *Overview* for a description of intervals.

This is the scripted version of the *gofor* element in interactive control.

Use this element to advance the solution. Specify the solution, display and storage intervals.

<advancefor>

<solutionint>

Solution interval goes here.

</solutionint>

<displayint>

Display interval goes here.

</displayint>

<storageint>

Storaage interval goes here.

</storageint>

</advancefor>

✓ Valid in *%tasks*

✓ See also

< advanceto >

✓ DTD

<!ELEMENT advancefor

(solutionint,

displayint?,

storageint?

)

>

<!ELEMENT solutionint (#PCDATA)>

<!ELEMENT displayint (#PCDATA)>

<!ELEMENT storageint (#PCDATA)>

The default *displayint* is *solutionint* / 10. The default *storageint* is *displayint*.

End

< advanceto >

See this volume's *Overview* for a description of intervals.

This is the scripted version of the *goto* element in interactive control.

Use this element to advance the solution. Specify the solution, display and storage intervals.

This is a little tricky, but it works great in tidying up the progress of a solution. A solution interval is specified, but the solution doesn't go the full interval. Instead, it goes until the independent variable (*System.X*) has increased to the next multiple of the specified solution interval.

Thus, this is a modified *advancefor* element.

<advanceto>

<solutionint>

Solution interval goes here.

</solutionint>

<displayint>

Display interval goes here.

</displayint>

<storageint>

Storaage interval goes here.

</storageint>

</advanceto>

As an example, suppose the independent variable has a current value of 2000. The Go menu has a goto menu item with a solution interval of 1440 (and maybe a label of *To Next Day*). When user clicks this choice, the solution will be advanced to the next multiple of 1440, which is 2880, for a net advance of 880.

✓ Valid in *%tasks*

✓ See also

< advancefor >

✓ DTD

<!ELEMENT advancetor

(solutionint,

displayint?,

storageint?

)

>

<!ELEMENT solutionint (#PCDATA)>

<!ELEMENT displayint (#PCDATA)>

<!ELEMENT storageint (#PCDATA)>

The default *displayint* is *solutionint* / 10. The default *storageint* is *displayint*.

End

< call >

Use this element to call or invoke a *block* of math in *definitions*.

The element is also used in definitions (see *Volume 2. Structure*, *Definitions*).

```
<call>
    Block name goes here.
</call>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT call (#PCDATA)>
```

End

< def >

Use this element to change the value of a variable.

This element is also used in *definitions* (see *Volume 2. Structure*, *Definitions*).

```
<def>
    <name>
        Variable name goes here.
    </name>
    <val>
        Mathematical expression goes here.
    </val>
</def>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT def (name, val)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT val (#PCDATA)>
```

End

< eval >

Use this element to implement conditional scrip execution.

The mathematical expression in *test* is evaluated. If it evaluates to TRUE, the tasks in the *truetasks* element are executed. If it evaluates to FALSE, the tasks in the *falsetasks* element are executed.

```
<eval>
    <test>
        Expression to be evaluated goes here.
    </test>
    <truetasks>
        Tasks for TRUE outcome go here.
    </truetasks>
    <falsetasks>
        Tasks for FALSE outcome go here.
    </falsetasks>
</eval>
```

This element is similar to the *if* element in *%math* (see *Volume 2. Structure*, *Definitions*).

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT eval

```
    (   test,
        truetasks?,
        falsetasks?
    )
>

<!ELEMENT test (#PCDATA)>
<!ELEMENT truetasks %tasks; >
<!ELEMENT falsetasks %tasks; >

End
```

< exitscript >

When this element is encountered, the script is terminated.

I don’t see how this can be used, but there may be an occasional and unanticipated use.

```
<exitscript/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT exitscript EMPTY>
```

End

< fileaccept >

Use this element to check the value of the named variable. The evaluation is recorded in the open file.

If the value of the variable is less than the minimum specified or greater that the maximum specified, a not accepted message is recorded.

This element is identical to *accept* except that the output device is a file rather than the screen.

```
<fileaccept>
    <name>
        Name of variable to be evaluated goes here.
    </name>
    <min>
        Minimum acceptable value goes here.
    </min>
    <max>
        Maximum acceptable value goes here.
    </max>
    By default, the script continues after logging an unacceptable value.
    But, you can use the empty element below to stop the script after
    an unacceptable value has been recorded.
    <stoponnotaccepted/>
</fileaccept>
```

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT fileaccept

(name,

min?,

max?,

stoponnotaccepted?

)

>

<!ELEMENT name (#PCDATA)>

<!ELEMENT min (#PCDATA)>

<!ELEMENT max (#PCDATA)>

<!ELEMENT stoponnotaccepted EMPTY>

If *min* is not specified, the lower bounds are not checked. If *max* is not specified, the upper bounds are not checked. If *stoponnotaccepted* is not specified, the script continues.

End

< fileannotate >

Use this element to add text to the open file.

```
<fileannotate>
    Annotation text goes here.
</fileannotate>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileannotate (#PCDATA)>
```

End

< fileclose >

Use this element to close the open file.

<fileclose/>

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT fileclose EMPTY>

End

< filenewline >

Use this element to write an empty line to the open file.

<filenewline/>

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT filenewline EMPTY>

End

< fileopenappend >

Use this element to open an existing file with the specified filename. Writes to the file will be appended to the end of the file.

```
<fileopenappend>
    Filename goes here.
</fileopenappend>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileopenappend (#PCDATA)>
```

End

< fileopencreate >

Use this element to create and open a new file with the specified filename.

```
<fileopencreate>
    Filename goes here.
</fileopencreate>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileopencreate (#PCDATA)>
```

End

< fileroster >

Use this element to identify the variables whose values will be written into the open file by solution tracking and *fileupdate*.

See *Volume 5. Misc*, *File Formats* for the file format.

```
<fileroster>
    <variable>
        <name>
            Variable's name goes here.

        </name>
        <format>
            Format details go here >.

        </format>
    </variable>
    More variable's go here.
</fileroster>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileroster (variable*)>
<!ELEMENT variable (name, format?)>

<!ELEMENT name (#PCDATA)>
```

<!ELEMENT format %format; >

End

< filestarttracking >

Tracking is the process of writing the values of selected variables into the open file at each display interval for all or part of a solution.

Variables, formats and file layout are specified by the *filelayout* element.

Tracking is started by the this element and stopped by *filestoptracking* element.

```
<filestarttracking/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT filestarttracking EMPTY>
```

End

< filestoptracking >

Tracking is the process of writing the values of selected variables into the open file at each display interval for all or part of a solution.

Variables, formats and file layout are specified by the *filelayout* element.

Tracking is started by the *filestarttracking* element and stopped by this element.

```
<filestoptracking/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT filestoptracking EMPTY)>
```

End

< filetimestamp >

Use this element to write the current time and date into the open file.

```
<filetimestamp/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT filetimestamp EMPTY>
```

End

< fileupdate >

Use this element to write the current values of selected variables into the open file. Variables, formats and file layout are specified by the *filelayout* element.

```
<fileupdate/>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileupdate EMPTY>
```

End

< fileverify >

Use this element to verify that the specified file exists. The script is terminated if the file does not exist.

```
<fileverify>
    Filename goes here.
</fileverify>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT fileverify (#PCDATA)>
```

End

< filewriteheader >

Use this element to write a header in the open file. Header is composed using the *label* specifications in the *filelayout* element.

```
<filewriteheader/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT filewriteheader EMPTY>
```

End

< loadics >

Load initial conditions from the specified initial conditions file.

```
<loadics>
    Existing IC's filename goes here.
</loadics>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT loadics (#PCDATA)>
```

End

< message >

Text is display in a message box when this element is encountered. Usually used in conditional code, but it might also be used to announce progress in a big script.

```
<message>
    Text for display goes here.
</message>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT message (#PCDATA)>
```

End

< pause >

When this element is encountered, the execution of the script pauses. User response is required. The user can then choose to continue the script or to exit the script.

```
<pause/>
```

- ✓ Valid in *%tasks*.
- ✓ DTD

```
<!ELEMENT pause EMPTY>
```

End

< repeatfor >

Execute the tasks in this element the number of times specified by *count* (base 1).

```
<repeatfor>
    <count>
        Number of repetitions goes here.
    </count>
    Tasks to be executed go here.
</repeatfor>
```

✓ Valid in *%tasks*

✓ DTD

```
<!ELEMENT repeatfor (count, %tasks; )>
<!ELEMENT count (#PCDATA)>
```

End

< repeatwhile >

Evaluate the mathematical expression in *test*. If the result is TRUE, execute the tasks in this element. Repeat the evaluation and execution sequence until the result is FALSE.

```
<repeatwhile>
    <test>
        Expression to be evaluated goes here.
    </test>
    Tasks to be executed go here.
</repeatwhile>
```

- ✓ Valid in *%tasks*
- ✓ DTD

```
<!ELEMENT repeatwhile (test, %tasks; )>
<!ELEMENT test (#PCDATA)>
```

< replacenote >

A note is text that is displayed on a panel and stored in an initial conditions file. It can be edited or replaced (as is done here) to individualize the initial conditions data set.

<replacenote>

<name>

Note's name goes here.

</name>

<line>

Line of text goes here.

</line>

More lines go here.

</replacenote>

✓ Valid in *%tasks*

✓ DTD

<!ELEMENT replacenote (name, line*)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT line (#PCDATA)>

End

< reset >

Variables have three different values.

The *current* value is the value that is seen as a solution advances. It can change continuously.

The *initial* value is the value at the start of a solution. *Restart* sets the variables to their initial values. But initial values are not totally invariant. Initial values are assigned new values when an initial conditions file is loaded. In addition fixed parameter values and locked parameter values are not changed during a restart.

The *original* value is the value when the model is first loaded. *Reset* sets the variables to their original values. This means that values from initial conditions files are discarded. Locked parameter values are unlocked. All variables, including fixed parameters, are set to their original values.

Use this element to reset the model.

```
<reset/>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT reset EMPTY>
```

End

< restart >

Variables have three different values.

The *current* value is the value that is seen as a solution advances. It can change continuously.

The *initial* value is the value at the start of a solution. *Restart* sets the variables to their initial values. But initial values are not totally invariant. Initial values are assigned new values when an initial conditions file is loaded. In addition fixed parameter values and locked parameter values are not changed during a restart.

The *original* value is the value when the model is first loaded. *Reset* sets the variables to their original values. This means that values from initial conditions files are discarded. Locked parameter values are unlocked. All variables, including fixed parameters, are set to their original values.

Use this element to restart the solution.

```
<restart/>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT restart EMPTY>
```

End

< saveics >

Use this element to save the initial condition values to the specified file.

```
<saveics>
    IC's filename goes here.
</saveics>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT saveics (#PCDATA)>
```

End

< selectpanel >

Use this element to change the panel being displayed. May be useful in a complex script when you want to watch what is going on.

<selectpanel>

Panel name goes here.

</selectpanel>

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT selectpanel (#PCDATA)>

End

< setdisplayspeed >

The display speed (or screen refresh rate) is timed to provide a smooth solution. A slow display speed is sometimes useful when giving a demonstration. A fast display speed is sometimes uses in a script when no one is watching.

Use this element to set the display speed.

```
<setdisplayspeed>
    Display speed goes here.
</setdisplayspeed>
```

Possible display speeds are

0 - Slowest

1 - Slow

2 - Normal

3 - Fast

4 - Fastest

✓ Valid in *%tasks*.

✓ DTD

<!ELEMENT setdisplayspeed (#PCDATA)>

End

< setfileattribute >

While this element may find more than one use, it was originally created to set initial condition files to read-only to prevent loss by overwriting.

<setfileattribute>

<filename>

Filename goes here.

</filename>

<attribute>

Attribute goes here.

</attribute>

</setfileattribute>

Possible attributes are NORMAL and READONLY.

- ✓ Valid in *%tasks*
- ✓ DTD

<!ELEMENT setfileattribute (filename, attribute)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT attribute (#PCDATA)>

End

< setpagerstatus >

Use this element to set the pager status. A page requires user response and this might not be desirable in some scripts.

```
<setpagerstatus>
    Pager status is ON or OFF.
</setpagerstatus>
```

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT setpagerstatus (#PCDATA)>
```

End

< setworkdir >

Use this element to set the working directory.

```
<setworkdir>
    Path to working directory goes here.
</setworkdir>
```

Working directory specification must contain a drive letter. An example: *C:\Scripts*

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT setworkdir (#PCDATA)>
```

End

< setworkdir >

Use this element to set the working directory.

```
<setworkdir>
    Path to working directory goes here.
</setworkdir>
```

Working directory specification must contain a drive letter. An example: *C:\Scripts*

✓ Valid in *%tasks*.

✓ DTD

```
<!ELEMENT setworkdir (#PCDATA)>
```

End

< scripted >

Use this element as a container element for all of the scripted control elements.

```
<scripted>

    Scripted control elements go here.

</scripted>
```

✓ Valid in < control >

✓ See also < interactive >
< remote >
< misc >

✓ DTD The elements in *scripted* may be defined in any order and amount.

```
<!ELEMENT scripted

    (   fixedparm
        | %tasks
    )*
>
End
```

Volume 4. Display

Common

Overview

The common element is a container element for display objects that are designed to be used generally or by more than one panel.

Main common items are

- The name of the panel to be displayed first.
- The font to be used in the panels.
- The panel tree. This is the structure used to organize the pop menus that list all of the model's panels.

There are also some visual objects that might be defined here for sharing or might be defined in a panel. These are the elements *maplist*, *repeatlist* and *symbol*. They are described in *Panel* since they are usually used more extensively in that element.

End

< branch >

Use this element to create a new pop-down menu or sub-menu.

The tree will be used to define one or more pop menus (with sub-menus) or a tree view of available panels.

```
<branch>
    <name> Name of new menu. </name>
    <label> Display this label. </label>
    <parent> Name of parent menu. </parent>
</branch>
```

The solver's main menu can be a parent. Its name is MAINMENU.

End

< common >

Use this element as a container element for common display elements.

<common> Common display elements. </common>

Some common display elements are *displayfirst*, *font* and *tree*.

End

< display >

Use this element as a container element for display elements. The main display elements are *common* and multiple *panel*'s.

<display> Common & panel elements. </display>

Put all the *common* elements before the *panel* elements to allow panels to refer to common display objects.

End

< displayfirst >

Use this element to identify the panel that should be displayed first.

<displayfirst> Display this panel 1st. </displayfirst>

If this element is not used, the first panel created will be the first panel displayed.

End

< font >

NOTE – This element has not yet been implemented. The font just now comes from the default values.

Use this element to define the font to be used in panels.

```
<font>
    <name> Font name. </name>
    <style> Font style. </style>
    <size> Font size (pts.). </size>
    <color> Font color. </color>
</font>
```

Default values for *name*, *style*, *size* and *color* are *Tahoma*, *regular*, *9* and *black*.

End

< leaf >

Use this element to place a panel on a pop-down menu or sub-menu.

```
<leaf>
    <name> Name of panel. </name>
    <label> Display this label. </label>
    <parent> Name of parent menu. </parent>
</leaf>
```

End

< separatorbar >

Use this element place a spacer in one of the pop-down panel menus.

```
<separatorbar>
    <parent> Name of menu. </parent>
</separatorbar>
```

End

< tree >

Use this element organize the display's panels in a tree structure.

The tree will be used to define one or more pop menus (with sub-menus) or a tree view of available panels.

<tree> Tree elements go here. </tree>

Tree elements are

- A panel with name, label and menu name (a leaf).
- A pop menu with name, label and parent menu name (a branch).
- A menu separator bar.

Pop menus can be hooked into the right end of the solver's main menu.

End

Panel

Overview

A panel is the solver's basic display element for simulation results. A panel contains numerical values, graphs, labels and other visual objects.

The panel element is used to specify all of this including panel layout.

The model builder has many different visual objects available to contribute to pleasing screen design. Thus, the following section is a bit on the bulky side.

End

< actionbutton >

An action button is a button containing an exclamation point (!). An optional label appears to the left or right of the button.

Click the button and the specified math block is executed.

Action buttons have been used to take blood samples, inject insulin and reset odometers.

Use the *actionbutton* element to create an action button in a panel.

```
<actionbutton>
    <row> Top. </row>

    <col> Left side. </col>

    <blockname> Execute on click. </blockname>


    <label> Label next to button. </label> or

    <nolabel/>


    <layout> LABELLEFT or LABELRIGHT. </layout>
</actionbutton>
```

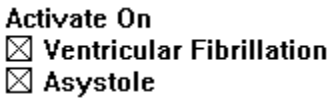
The default *label* is none. The default *layout* is LABELRIGHT.

End

< checkbox >

Several commands allow the user to change the value of one of the model's parameters.

The *checkbox* element draws a box and shows a check if the parameter has a non-zero value. The box is empty when the parameter is zero.



User clicks on box to toggle between checked (1) and not checked (0) values.

Use this element to create a checkbox.

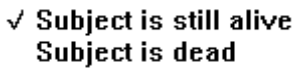
```
<checkbox>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Parameter name. </name>

    <label> Label (Optional). </label>

</checkbox>
```

The default *label* is the parameter name.

End

< checkmark >

Use this element to display a checkmark as a function of the numerical value of a variable.



If the value of the variable is zero, no checkmark is displayed (bottom line above). If the value is not zero, a checkmark is displayed (top line above).

```
<checkmark>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Variable name. </name>

    <label> Label (Optional). </label>

    <layout> CHECKLEFT or CHECKRIGHT. </layout>

</checkmark>
```

If a label is not specified, the variable name is used as a label. The default *layout* is CHECKLEFT.

End

< editbox >

Several elements allow the user to change the value of one of the model's parameters.

The *editbox* element puts a box, a button, a label and the parameter's current value on the screen. The user can enter a new value in the box and then click the button to apply the value to the parameter.

```
<editbox>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Parameter name. </name>

    <label> Label (Optional). </label>

    <min> Minimum value allowed. </min>

    <max> Maximum value allowed. </max>

</editbox>
```

If the *label* is not specified, the parameter's name is used as a label.

If *min* and/or *max* are not specified, the new value is not restricted in that respective direction.

End

< format >

The *format* element is not a top-level element in panels and group boxes. It is used a bit further down. But it is such a wordy element that I wanted to document it just once – right here.

Use this child element to specify a format to be used when a numerical value is displayed.

Specify just one of the several possible formats.

```
<format>
    <none/> or
    <integer/> or
    <decimal> Digits after decimal. </decimal> or
    <list> List name. </list> or
    <timer/> or
    <timestamp/> or
    <boolean/>

    <fieldwidth> Field width in columns. </fieldwidth>
    <justify> LEFT, CENTER or RIGHT. </justify>
</format>
```

If the format element is not used, the number is not formatted. This means that the image will probably have lots of digits to the right of the decimal point.

The default *fieldwidth* is 12 columns. The default *justify* is LEFT.

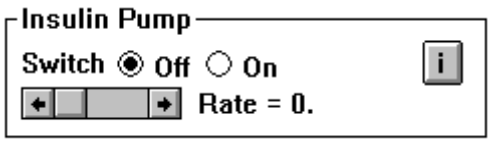
The *list* format requires some explanation. A list is an ordered set of numerical value and image pairs. The list format takes a variable's

numerical value, finds the closest numerical value in the list, and then displays the corresponding image. See *maplist* and *repeatlist*.

End

< groupbox >

Use the *groupbox* element to draw a *groupbox* with a title. The title can be used to identify the theme of the box's contents.



<groupbox>

<row> Top. </row>

<col> Left side </col>

<high> Rows high </high>

<wide> Columns wide </wide>

<title> Title (Near upper left) </title>

The box's visual objects are defined here.

</groupbox>

The row, col, high and wide elements are floating point.

When a groupbox is opened, the drawing origin (row = 0.0, col = 0.0) is moved to the top left corner of the groupbox. When the groupbox is closed, the drawing origin is returned to its previous position.

If a title is not specified, the box has no title. It appears as a simple rectangle rather than the titled rectangle shown above.

End

< infobutton >

This element creates a button showing the letter *i* with an optional label located to the left or right of the button.

When the button is clicked, the specified lines of text (info) are displayed in a message box.

```
<infobutton>
    <row> Row. </row>

    <col> Column (Left side). </col>


    <label> Label (Optional). </label> or

    <nolabel/>


    <layout> LABELLEFT or LABELRIGHT. </layout>

    <line> A line of info. </line>


    More lines of info go here.


</infobutton>
```

The default label is no label. The default *layout* is LABELRIGHT.

End

< label >

Use the *label* element to create a label.

```
<label>
    <row> Row. </row>
    <col> Column (Left side). </col>
    <text> Label's text. </text>
    <fieldwidth> Columns wide. </fieldwidth>
    <justify> LEFT, CENTER or RIGHT. </justify>
    <color> Text color. </color>
</label>
```

The default *fieldwidth* is that width is adjusted automatically. Specify field width only when doing some fancy center or right justification.

Default *justify* is LEFT. Default *color* is BLACK.

End

< maplist >

Lists are ordered pairs of numerical values and text.

For example, this data might be used to define heart strength:

Numerical Value	Text
0.0	None
1.0	Low
2.0	Normal
3.0	High

Lists are used to define qualitative values and to limit choices during a parameter change.

Use this element to create a list that defines qualitative values.

```
<maplist>
    <name> Unique list name. </name>

    <map>
        <val> Value. </val> <img> Image. </img>
    </map>

    More map elements go here.
```

```
</maplist>
```

As an example, the list created below represents the map shown above.

```
<maplist>
    <name> HeartStrength </name>

    <map><val> 0.0 </val><img> None </img></map>
    <map><val> 1.0 </val><img> Low </img></map>
    <map><val> 2.0 </val><img> Normal </img></map>
    <map><val> 3.0 </val><img> High </img></map>

</maplist>
```

End

< panel >

Use the panel element as a container element for a panel's visual objects.

A panel has a name and an optional tab label. The name is unique (and hopefully it is quite descriptive). The optional tab label replaces the full name (if it is too cumbersome) in the tab above the active panel (this is a future development).

```
<panel>
    <name> Panel name. </name>
    <tablabel> Panel's tab label. </tablabel>

    Panel's visual objects are defined here.

</panel>
```

The default *tablabel* is the panel *name*.

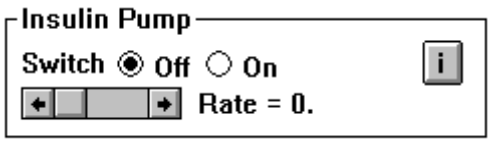
There is no limit on the number of panels that can be defined.

End

< radiobuttons >

Several elements allow the user to change the value of one of the model's parameters.

The radiobuttons element creates some radio buttons which can be used to select a new value for a parameter.



<radiobuttons>
<row> Row. </row>

<col> Column (Left side). </col>

<name> Parameter name. </name>

<listname> Button labels. </listname>

<label> Label (Optional). </label> or

<nolabel/>

<layout> HORZ or VERT. </layout>

</radiobuttons>

The named list above is the mapped list that supplies radiobutton labels and parameter values, such as Off means 0 and On means 1.

If the label is not specified, the parameter's name is used as the label.

Layout specifies button layout. The two possible layouts are horizontal (HORZ) and vertical (VERT). The default value is HORZ.

End

< repeatlist >

Lists are ordered pairs of numerical values and text.

Lists can be used to limit choices during a parameter change.

The *repeatlist* element produces a list of evenly spaced numerical values. The values and their images are identical.

```
<repeatlist>
    <name> Unique list name. </name>

    <firstval> First value in list. </firstval>


    <repeat>

        <reps> Repetitions @ step size. </reps>

        <stepsize> Step size. </stepsize>

    </repeat>


    More repeat elements go here.


</repeatlist>
```

The first value in the list and the step size are floating point. The default first value is 0.0.

Repetitions is an integer.

As an example, the list created below steps from 1 to 10 in increments of 1 and then steps from 10 to 100 in increments of 10.

```
<repeatlist>
    <name> Example </name>
    <firstval> 1.0 </firstval>

    <repeat>
        <reps> 9 </reps><stepsize> 1 </stepsize>
    </repeat>

    <repeat>
        <reps> 9 </reps><stepsize> 10 </stepsize>
    </repeat>

</repeatlist>
```

End

< scale >

The *scale* element is like the *format* element. It is not a top-level element in panels and group boxes. But I'll document it here just once to avoid repetition.

Scale is a child element that is used to specify the scale used on the axes of graphs. It is applied to the x- and y-axes of graphs and to the only axis of *bargraph*s and *valuebar*s.

```
<scale>
    <min> Minimum scale initial value. </min>

    <max> Maximum scale initial value. </max>

    <inc> Auto-scaling increment. </inc>

</scale>
```

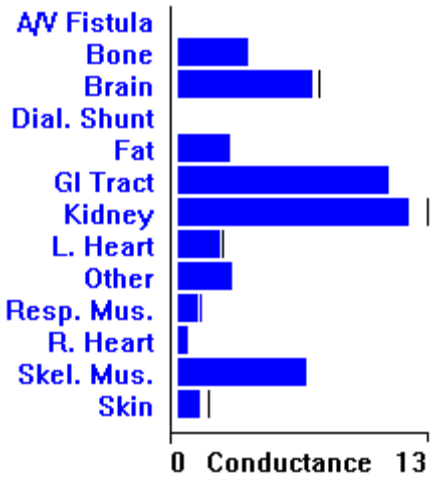
Min, *max* and *inc* are floating point.

The default value for *inc* is *max* minus *min*.

End

< showbargraph >

Use the showbargraph element to create a horizontal bar graph for one or more variables.




```
<showbargraph>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <high> Rows high </high>

    <wide> Columns wide </wide>

    <leftmargin> Margin columns wide </leftmargin>

    <title> Label alone bottom. </title>

    <showinitialvalues/>


    <bar>

        <name> Variable name. </name>
```

<label> Label (Optional). </label> or

<nolabel/>

<color> Bar and label color. </color>

<label> Label (Optional). </label>

</bar>

Additional bars go here. Adjust high accordingly.

<scale> See <scale>. </scale>

</showbargraph>

There are some default values for high and wide. Don't use the defaults – the results will be ugly.

The *leftmargin* element reserves some room for the bar labels.

The default *title* is no title.

The *showinitialvalues* empty element instructs the bar graph to draw a thin line at the level of the initial values. This marking is visible in the figure above. The default value is to not show initial values.

If a bar label is not specified, the variable's name is used as a label.

Default bar and label color is BLACK. I like BLUE.

End

< showbitmap >

Use the *showbitmap* element to load a bitmap from the application's resource and display it in a panel at the specified location.

```
<showbitmap>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Bitmap name. </name>

</showbitmap>
```

Bitmaps are given a name when they are stored in the application's resource. The trick is to know what these names are.

One work around is to look at HUMMOD's panels. When you see a bitmap that you like, look in the appropriate DES file for the name of the bitmap.

End

< showbitmapfile >

Use the *showbitmapfile* element to load a bitmap from a file and display it in a panel at the specified location.

```
<showbitmapfile>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Bitmap file name. </name>

</showbitmapfile>
```

End

< showclock >

Use this element to display the value of a variable using a digital clock format.

12:31 PM Day 7 Sec 0 Mon Day 8

```
<showclock>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Variable name. </name>

    <timebase> Variable's time base. </timebase>


    <ampm/>

    <secs/>

    <days/>

</showclock>
```

The available time bases are MILLISEC, SEC, MIN, HOUR, DAY, WEEK and MONTH

Use *ampm* empty element below to request AM|PM display. The default is 24 hour (international) time.

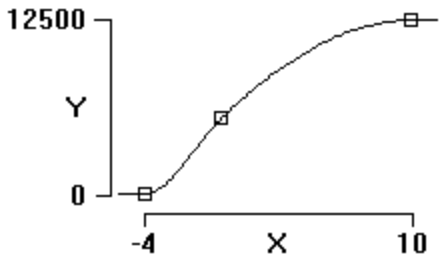
Use *secs* empty element below to request a display of seconds. The default is HH:MM only.

Use *days* empty element below to request a display of days. The default is no days.

End

< showcurve >

Use the *showcurve* element to create a graph of the output of a curve function (y-axis) as a function of the input (x-axis) over the full range of the curve's input. Put the graph at the current drawing point.



Curves can misbehave quite badly between points, so it is a good idea to view the total curve to see if it looks OK.

<showcurve>
<row> Row. </row>

<col> Column (Left side). </col>

<high> Rows high </high>

<wide> Columns wide </wide>

<leftmargin> Margin columns wide </leftmargin>

<xaxis>

<name> Variable name. </name>

<label> Label (Optional). </label> or

<nolabel/>

<scale> See <scale>. </scale>

```
        </xaxis>

        <yaxis>
            <label> Label (Optional). </label> or
            <nolabel/>

            <linetype> Line type. </linetype>
            <linecolor> Line color. </linecolor>
            <symbolname> Symbol. </symbolname>
            <scale> See <scale>. </scale>
        </yaxis>

        <curvename> Curve name. </curvename>

        <noghost> or

        <ghost>
            <linetype> Line type. </linetype>
            <linecolor> Line color. </linecolor>
            <symbolname> Symbol. </symbolname>
        </ghost>

    </showcurve>
```

The default value for *high* is 7 rows. The default value for *wide* is 30 columns.

The *leftmargin* element reserves some room for the bar labels. The default value for left margin is 6 columns.

Default line type is solid. Default line color is BLACK for graphs and GREY for ghosts.

A ghost will show the original curve shape and symbol location. This is probably of no value when the curve is not changing (for right now – always).

End

< showfile >

Use the *showfile* element to display the contents of a text file in a scrolling window.

```
<showfile>
    <row> Row (Top). </row>

    <col> Column (Left side). </col>

    <high> Rows high </high>

    <wide> Columns wide </wide>

    <name> File name. </name>

</showfile>
```

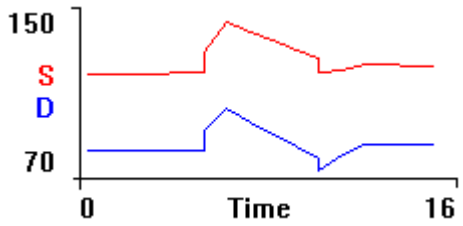
If filename is not specified, this element will display the file holding the model's root element.

The default window height is 20 rows. The default window width is 50 columns.

End

< showgraph >

Use the *showgraph* element to create a graph. One or more variables may be shown on the y-axis, sharing a common x-axis.



This is the graphing workhorse.

```
<showgraph>
    <row> Row (Top). </row>

    <col> Column (Left side). </col>

    <high> Rows high </high>

    <wide> Columns wide </wide>

    <leftmargin> Margin columns wide </leftmargin>


    <xaxis>

        <name> Variable name. </name>


        <label> Label (Optional). </label> or

        <nolabel/>
```

<scale> See <scale>. </scale>

</xaxis>

<yaxis>

<yvar>

<label> Label (Optional). </label> or

<nolabel/>

<linetype> Line type. </linetype>

<linecolor> Line color. </linecolor>

<symbolname> Symbol. </symbolname>

</yvar>

Additional Y variables go here.

<scale> See <scale>. </scale>

</yaxis>

</showgraph>

The default graph height is 7 rows. The default graph width is 30 columns. The default left margin is 6 columns.

If a label is not specified, the variable's name is used as a label.

The default linetype is SOLID. The default linecolor is BLACK.

If a symbol is not specified, no symbol is shown.

End

< showmap >

A map is used as a multiple-input, multiple-output function. We use the *copy* element to pass values to a block, we execute the block, and we use the *copy* element again to get the return values.

We can show a map roughly like we can show a curve (limiting the map to a single input variable and single output variable).

```
<showmap>
    <row> Row (Top). </row>

    <col> Column (Left side). </col>

    <high> Rows high </high>

    <wide> Columns wide </wide>

    <leftmargin> Margin columns wide </leftmargin>


    <xaxis>

        <name> Variable name. </name>


        <label> Label (Optional). </label> or

        <nolabel/>


        <scale> See <scale>. </scale>

    </xaxis>


    <yaxis>

        <name> Variable name. </name>
```

<label> Label (Optional). </label> or

<nolabel/>

<linetype> Line type. </linetype>

<linecolor> Line color. </linecolor>

<symbolname> Symbol. </symbolname>

<scale> See <scale>. </scale>

</yaxis>

<blockname> Map block name. </blockname>

<noghost> or

<ghost>

<linetype> Line type. </linetype>

<linecolor> Line color. </linecolor>

<symbolname> Symbol. </symbolname>

</ghost>

</showmap>

The default value for *high* is 7 rows. The default value for *wide* is 30 columns.

The *leftmargin* element reserves some room for the bar labels. The default value for left margin is 6 columns.

Default line type is solid. Default line color is BLACK for graphs and GREY for ghosts.

A ghost will show the original map result and symbol location. This is probably of no value when the map is not changing, but maps can change.

End

< showpanelname >

Use this element to display a panel's name in the panel.

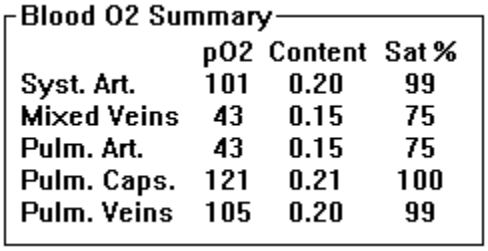
A typical location is the upper, left-hand corner of the panel.

```
<showpanelname>
    <row> Row. </row>

    <col> Column (Left side). </col>

</showpanelname>
```

End

< showtable >

The values of related variables can be displayed in a tabular format with labels located where needed.



Blood O2 Summary

	pO2	Content	Sat %
Syst. Art.	101	0.20	99
Mixed Veins	43	0.15	75
Pulm. Art.	43	0.15	75
Pulm. Caps.	121	0.21	100
Pulm. Veins	105	0.20	99

Use the *showtable* element to create a table.

A table is one or more side-by-side columns. Each column contains a stack of cells. Each cell displays

- A numerical value, or
- Text, or
- Nothing.

```
<showtable>
    <row> Row (Top). </row>

    <col> Column (Left side). </col>


    <column>

        <indent> Indent columns. </indent>

        <fieldwidth> Field width columns. </fieldwidth>
```

<justify> LEFT, CENTER or RIGHT. </justify>

<valuecell>

<name> Variable name. </name>

<format> See <format>. </format>

</valuecell>

or

<textcell> Text. </textcell>

or

<emptycell/>

More cells go here.

</column>

More columns go here.

</showtable>

End

< showvalue >

Use the *showvalue* element to display a variable's formatted numerical value.

Temperature = 98.5

This is a display workhorse.

```
<showvalue>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <name> Variable name. </name>

    <format> See <format>. </format>


    <label> Label (Optional). </label> or

    <nolabel/>


    <noequals/>

    <color> Text color. </color>
</showvalue>
```

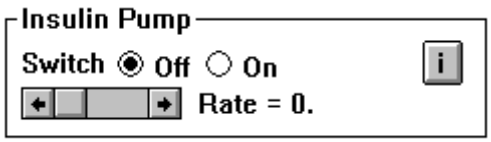
If the label is not specified, the variable name is used as a label. The default color is BLACK.

End

< slidebar >

Several elements allow the user to change the value of one of the model's parameters.

The *slidebar* element creates a slider that can be used to select a new value for the specified parameter.



A list is used to map the position of the slide to the parameter's values.

```
<slidebar>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <high> Rows high </high>

    <wide> Columns wide </wide>

    <name> Variable name. </name>

    <listname> List name. </listname>


    <label> Label (Optional). </label> or

    <nolabel/>


    <fieldwidth> Field width. </fieldwidth>

</slidebar>
```

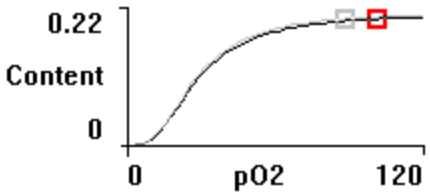
If a label is not specified, the name of the parameter is used as a label. The default height is 1.0 row. The default width is 12 columns.

Sometimes the slidebar width plus the label plus the field width for the value is too wide for the containing group box. This visual draw right through the right side of the group box. You can correct this situation by decreasing the width of the slidebar, using a less verbose label or decreasing the numerical field width (my preference).

End

< symbol >

A symbol is a little box or circle or X that is used to represent a point on a graph.



Use the *symbol* element to create a symbol.

```
<symbol>
    <name> Unique symbol name. </name>

    <style> BOX, CIRCLE or X. </style>

    <color> Line color. </color>

    <size> Symbol size. </size>

    <linewidth> Line width. </linewidth>

</symbol>
```

The default *color* is BLACK. The default *size* is 3. The default *linewidth* is 2.

End

< valuebar >

A value bar is a horizontal bar with minimum and maximum scales at the ends and a symbol somewhere along the bar showing the current value of the variable.

Use the *valuebar* element to create a value bar.

```
<valuebar>
    <row> Row. </row>

    <col> Column (Left side). </col>

    <wide> Columns wide </wide>

    <name> Variable name. </name>


    <label> Label (Optional). </label> or

    <nolabel/>


    <margins> Margins (Both sides). </margins>

    <symbolname> Symbol. </symbolname>

    <ghostname> Ghost. </ghostname>

    <scale> See <scale>. </scale>

</valuebar>
```

Margins reserve some room on either end for the scale values.

Default value for *wide* is 30. Default value for *margins* is 6. Default value for *label* is no label. Default name for *ghostname* is no ghost.

End

Volume 5. Misc

Overview

Volume 5. Misc covers some miscellaneous topics.

There is a lot of stuff that might go in here, but right now it appears that the topics are file formats (schema) for some of HUMMOD's XML I/O.

- HUMMOD Initialization. Application initialization data is stored in an initialization file and read at application startup. Current values are written to the file at application shutdown.

- Model Initial Conditions. The values of all model variables may be saved to a file and later reloaded as solution initial conditions.

- Model Solution. All the values of all model variables may be saved to a file as a solution image and later reloaded as a solution reconstruction.

Some additional file formats (XML schema) will probably be specified in the future. Candidates include

- Scripted Control. Scripted control is specified by an XML schema and parsed into the scripted controller.

- Script Log File. If requested, the progress of scripted control will be written to a log file as script processing proceeds.

- Interprocess Communication. These files implement communication between the solution controller and the equation solver (i.e. named pipes).

The basic idea here is that all data transport will be implemented in XML.

End

HUMMOD Initialization

Application initialization data is stored in an initialization file and read at application startup.

The file base name is the same as the application file base name. The file extension is *INI*.

Current data is written back to the file at application shutdown.

The document element is *ini*.

```
<ini>

    <time> Current time logged </time>

    <window>
        <top> Window top pixels </top>
        <left> Window left pixels </left>
        <high> Window high pixels </high>
        <wide> Window wide pixels </wide>
    </window>

    <setxtoic/> Set initial X in IC's

    <arrows> Panels remembered </arrows>

</ini>
```

A typical INI data set is shown below.

```
<ini>

    <time> Thu Nov 13 08:04:19 2008 </time>

    <window>

            <top> 16 </top>

            <left> 209 </left>

            <high> 603 </high>

            <wide> 516 </wide>
```

```
        </window>
        <arrows> 6 </arrows>
    </ini>

End
```

Model Initial Conditions

The values of all model variables may be saved to a file and later reloaded as solution initial conditions.

Save and load are *File* menu selections.

The usual file extension is *ICS*.

The document element is *ics*.

```
<ics>

    <time> Current time logged </time>

    <var>
        <name> Variable name </name>
        <val> Variable value </val>
        <state> Variable state </state>
    </var>

    More variables go here. Variable state is used by the timer
    variables. It is optional with a default value of 0.

    <note>
        <name> Note name </name>
        <text> Note text </test>
    </note>

    More notes go here.

</ics>
```

A fragment of a typical ICS data set is shown below.

```
<ics>


…

<var>
    <name> RightVentricle-Systole.A </name>
    <val> 3.52322260854839 </val>
```

```
    </var>

    <var>
        <name> RightVentricle-Systole.A-Basic </name>
        <val> 3.53 </val>
    </var>

    …

    </ics>
```

End

Model Solution

All the values of all model variables may be saved to a file and later reloaded as a solution image.

Save and load are *File* menu selections.

The usual file extension is *SOLN*.

The document element is *solution*.

```
<solution>

    <time> Current time logged </time>

    <index> Base-0 storage count </index>

    <var>
        <name> Variable name </name>
        <val> Variable value </val>

        All values for a variable are stored together in an implicitly
        indexed cluster. So, more solution values go here.

        <state> Variable state </state>
    </var>

    More variables go here.

    Variable state is used by the timer variables. It is optional with a
    default value of 0. Note that only the current (final) value of the
    state is stored in this document.

</solution>
```

A fragment of a typical SOLN data set is shown below.

```
<solution>
<time> Thu Nov 13 08:05:37 2008 </time>

<index> 20 </index>

…
```

<var>
<name> SystemicVeins.Inflow </name>
<val> 4211.37268565819 </val>
<val> 4209.973966819 </val>
<val> 4210.43347275471 </val>

… Total value count is base-0 index.

<val> 4217.87065793158 </val>
</var>

More variables go here.

</solution>

End