Introduction to AVS/Express Module Writing

PART I
Introduction

Objects

- Primitives
  - lowest level objects available
  - boolean, int, float, double, string, pointer, prim
- Groups
  - composite object containing:
  - groups
  - primitives
  - methods

Modules and Macros

- A module is a group object containing:
  - primitives and groups (high-level data) objects
  - at least one method
- A macro is a high-level object, containing other objects:
  - modules
  - primitive and high-level data objects
  - more macros
  - connections between those objects
  - visual representation, i.e., a network

Types of Modules

- V only
  - modules are implemented as V statements, using simple functions
  - the parameters themselves contain simple V expressions
- C method
  - method is implemented as a C function
  - use OM calls to access objects within module by name

Types of Modules (2)

- C++ method
  - method is implemented as a C++ class
  - class member objects are used to access module objects
  - use OMX calls for further object access and manipulation
- Fortran method
  - method is implemented as a Fortran subroutine
  - use OMF calls for object access

Types of Modules (3)

- Combination of the above types
  - some parameters may use V expressions
  - others may be set by executed methods
  - methods can be mixed

V Code

- Definitional Language
- Not a programming language!
- Defines all objects within Express environment
  - define methods and data objects for a module
  - define objects within a macro, and their connectivity
- V is generated by Express
- V may be written by user

Hierarchical Organization

- V aids the object oriented approach
- Allows easy reuse of existing components
- Express relies heavily on the hierarchical model
- Many objects in Express are built from many smaller objects
- Visual representation for the most basic components

Libraries

- Organization of objects into sensible groups
- Libraries themselves are Express objects
- New objects are placed into libraries
- V files define the libraries
- User may modify existing libraries
- Objects may be present in more than one library

Templates and Network Editor

- The Templates library contains *all* objects
  - organized into sets of libraries
- All other libraries and their objects, are either:
  - instances of the objects found in Templates
  - or Network Editor links (NElinks) to those objects

Methods

- The functional objects within Express
- Implemented in C, C++ or Fortran
- Are assigned parameters
  - which the method may read and/or write
  - which may trigger the method to be executed
  - which must be present and valid before method can execute
- Method has a unique name within the scope of the module

Attributes

- Parameters may be assigned attributes
  - read - method will read this parameter
  - write - method will write to this parameter
  - notify - method is notified of changes to parameter
  - required - method will not execute unless this parameter is present
- Assign to the parameter itself
- Assign to the parameter per method

Groups

- A composite object containing other objects
  - Similar to a data structure in C/C++
- A module is essentially a group object which contains at least one method
- High-level datatypes such as *Field* are hierarchical group objects
- Used to define a parameter block for a module

User Interface

- Express contains objects specifically for generating the user interface
- UI objects are combined to form macros
- Take data objects as parameters for widgets
- Widget manipulation used to change data objects
  - change of data is also reflected in widgets
- Parameter block common to UI and modules

PART II
Generating A Module

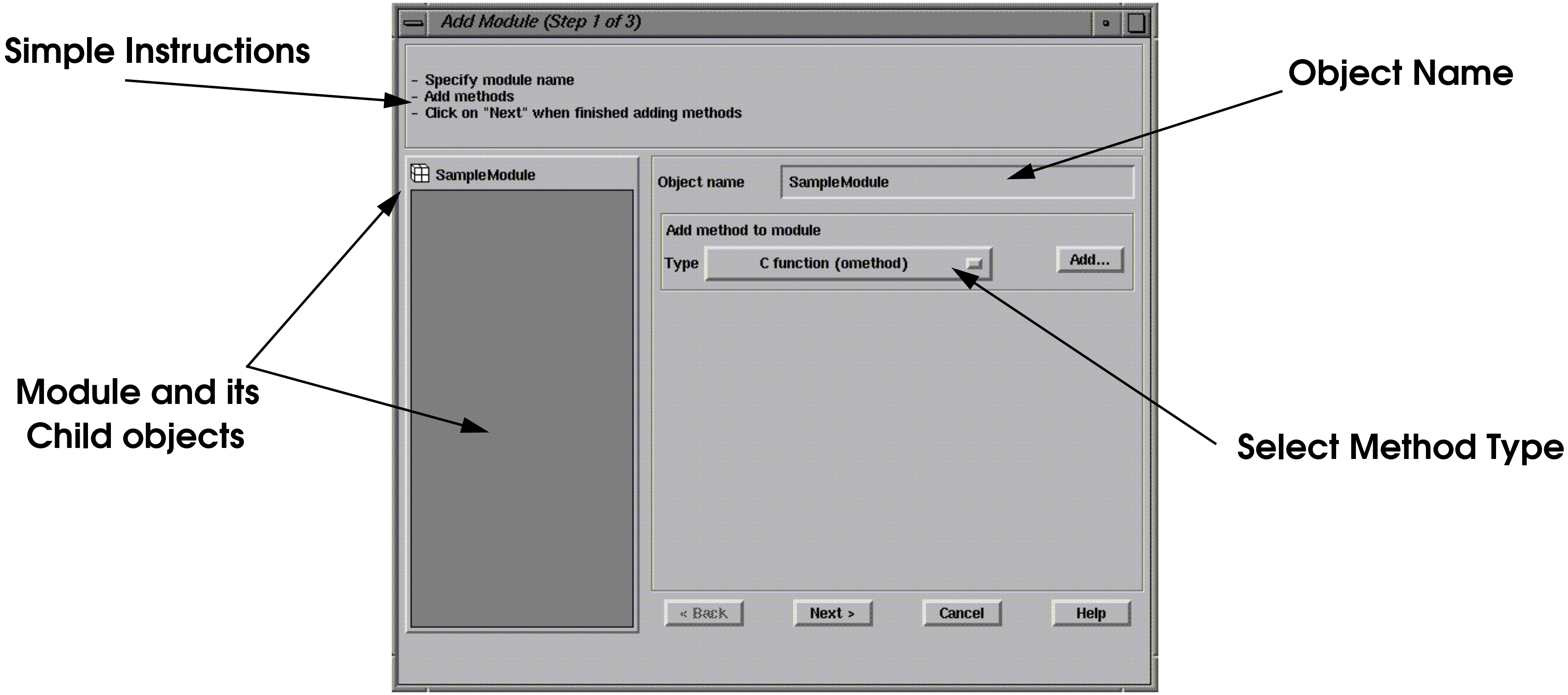
Approach

- Identify the following requirements:
  - a set of parameters (IO)
  - at least one method
  - choice of implementation of method(s)
- Adding the Module
  - drag and drop in Network Editor
  - Use the Add Module Tool
  - writing V code definitions

Add Module Tool

- Start the Scratch Pad Application
  - select *File -> New Application*
  - select Scratch Pad application
  - this allows objects to be built, without being executed
- Start the Add Module Tool
  - first select the Scratch Pad application, so that it is the current object
  - select *Object -> Add Module*
  - this is a step-by-step tool

Add Module


Simple Instructions
Object Name
Module and its
Child objects
Select Method Type
Add Module (Step 1 of 3)
- Specify module name
- Add methods
- Click on "Next" when finished adding methods
SampleModule
Object name
SampleModule
Add method to module
Type
C function (omethod)
Add...
« Back
Next >
Cancel
Help

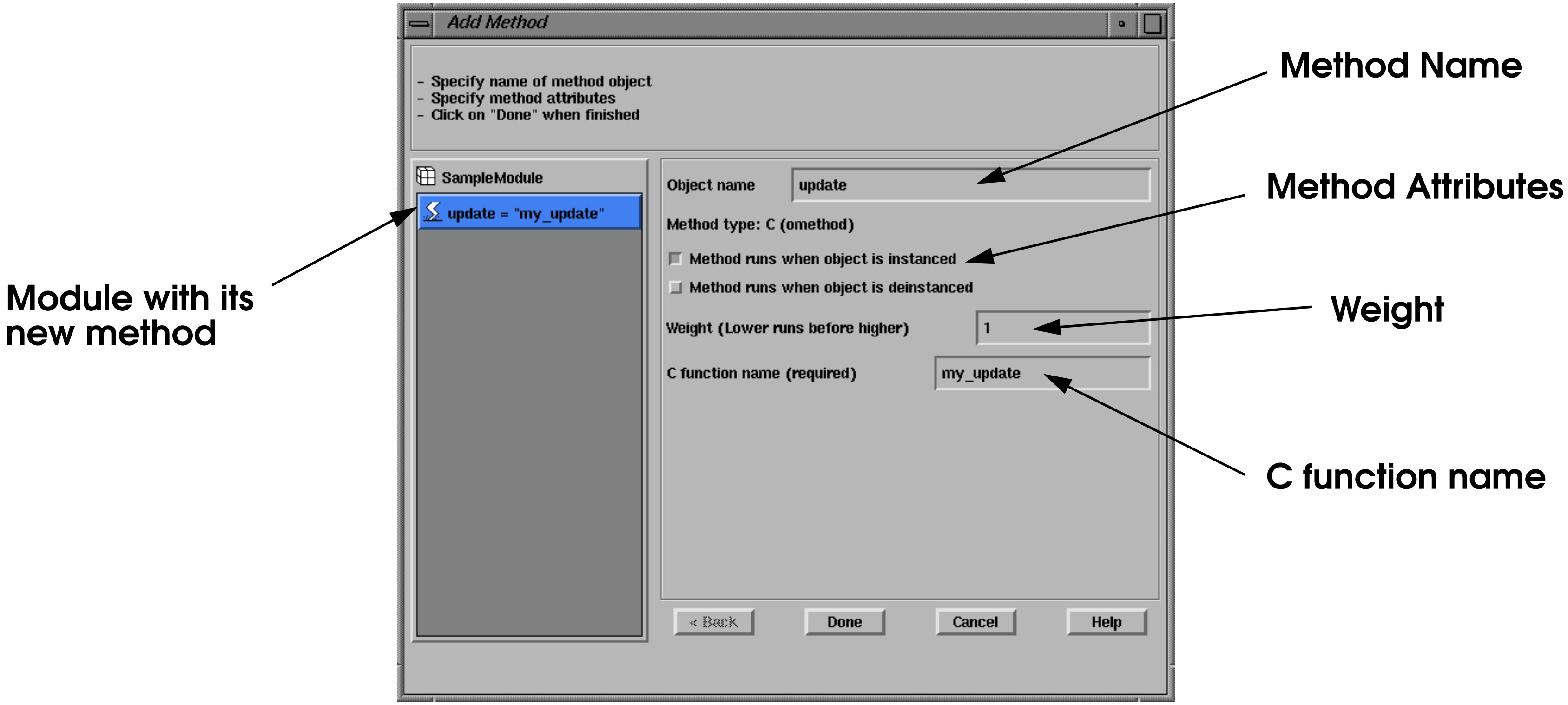
Add Module (2)

- (Step 1 of 3)
  - corresponds to step shown in Add Module Tool window
- Name the new module
  - change it from the default *NewModule*
  - press return to confirm
- Add methods to module
  - planning of methods
  - design already in mind

Add Module (3)

- (Step 1 of 3)
- Select type of method from menu
  - C++ member function (cxxmethod)
  - C function (omethod)
  - Fortran subroutine (fmethod)
- Then press the *Add...* button

Add Method


Add Method
- Specify name of method object
- Specify method attributes
- Click on "Done" when finished
SampleModule
update = "my_update"
Object name
update
Method type: C (omethod)
Method runs when object is instanced
Method runs when object is deinstanced
Weight (Lower runs before higher)
1
C function name (required)
my_update
« Back
Done
Cancel
Help
Method Name
Method Attributes
Weight
C function name
Module with its new method

Add Method (2)

- Name the method
  - *"update"* is common, but should be avoided
  - also displays previously selected method type
- Select certain attributes for the module
  - instanced - when the module is first dropped, it will execute this method
  - may execute more than one method
  - deinstanced - similarly, when the module is destroyed, the method is executed
  - used to clear up any internal data structures

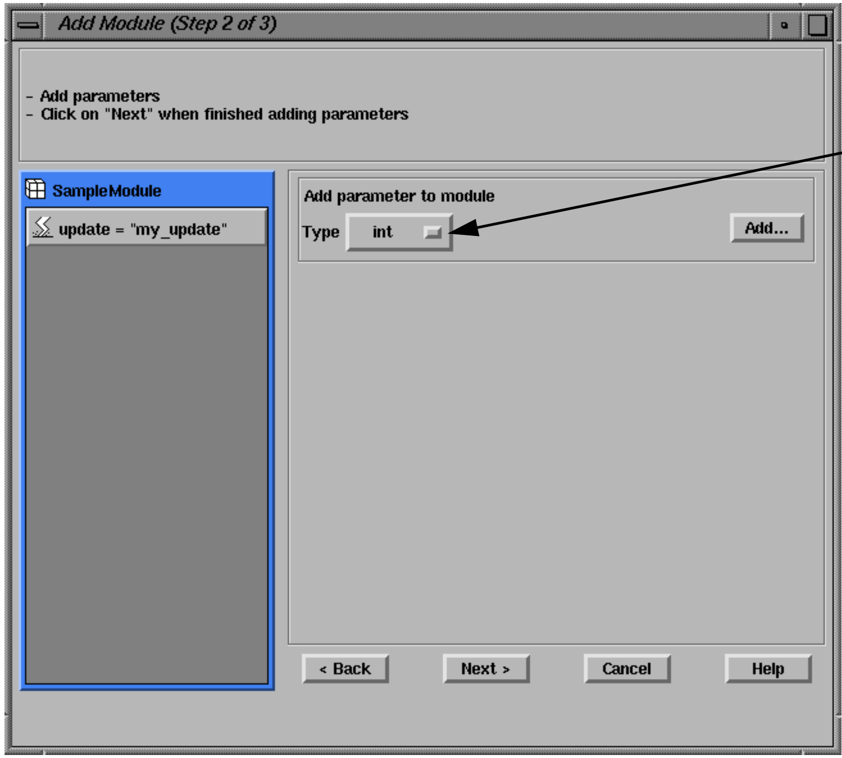
Add Method (3)

- Selecting a Weight value
  - multiple methods can execute when parameters change
  - lower weight values force methods to be executed before higher weight values
  - equal weights results in Express choosing which to execute first
- C Function Name
  - method is implemented as a C function
  - function will perform the actual task required
  - type the name of the function to be called; this is required
  - press *Done*

Add Module (4)

- (Step 1 of 3)
- Opportunity now to add more methods
- Repeat until all required methods have been added
- May add more later
  - use Object Editor Tool
  - edit V code directly
  - drag and drop methods from Standard Objects library
- When done, select *Next >*

Add Module (5)

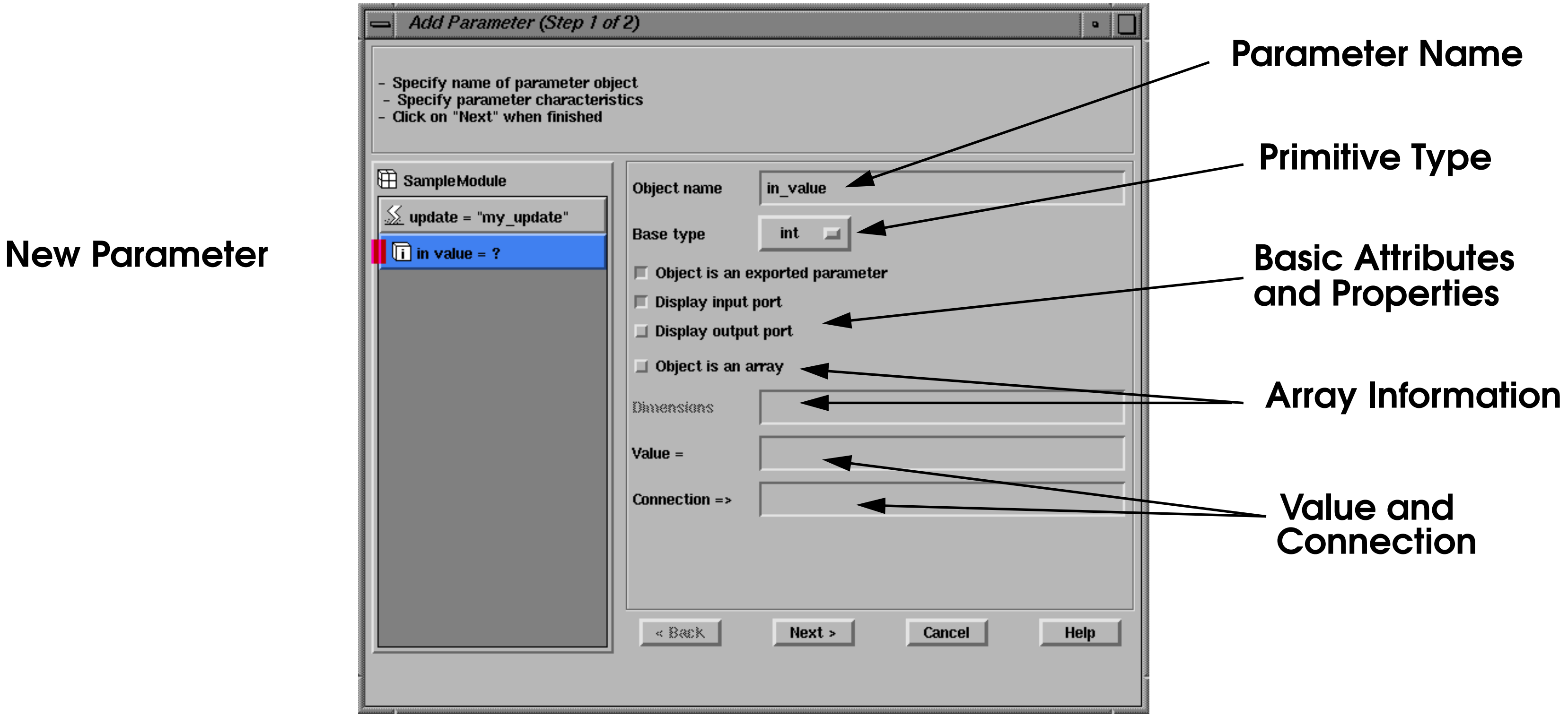


Select type of Parameter

Add Module (6)

- (Step 2 of 3)
- Now add parameters to module
- Select type from menu, choose from
  - boolean, char, byte, double, float, group, int, short, string and other
- These are all primitive data types
  - ‘other’ allows high-level data types to be selected
  - for example, Node_Data, Grid or user-defined groups
- Once selected, press *Add...*

Add Parameter



Add Parameter (2)

- (Step 1 of 2)
- Name the parameter
- Base type is shown, and may be changed here
- Select basic attributes
  - exported - parameter may be seen in the Network Editor
  - input port - a visible connection for input into parameter
  - output port - a visible connection for output of parameter
  - array - toggle on/off to make parameter an array of type chosen
- More later about arrays

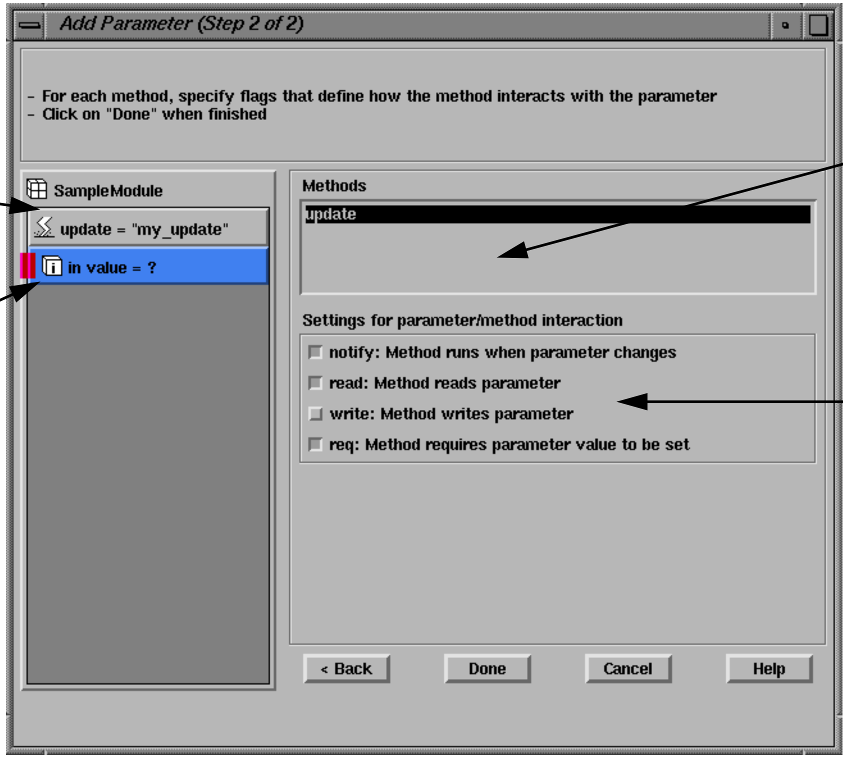
Add Parameter (3)

- (Step 1 of 2)
- Parameter may be given an initial value
- Parameter may be connected to another object by name
- More later about connections
- Press *Next >* when done

Add Parameter (4)

Module and child objects

Current Parameter



List of Methods

Method/Parameter Attributes

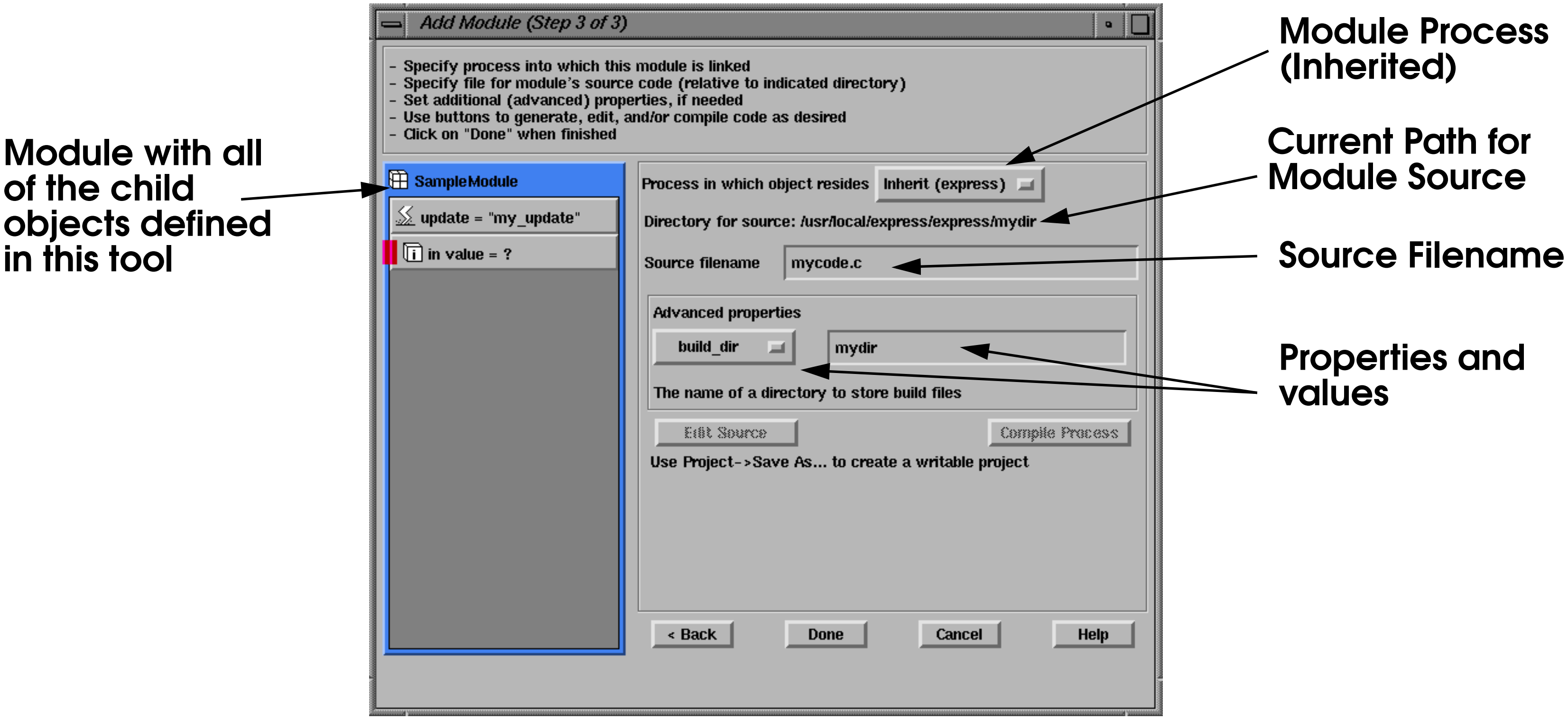
Add Parameter (5)

- (Step 2 of 2)
- Now assign how each method uses the new parameter
- All methods are listed in the *Methods* list
- Below are four attributes for that method/parameter pairing
  - notify - method will execute when parameter changes
  - read - method will read the value of the parameter
  - write - method will write new values to the parameter
  - req - parameter must have a valid value before method will execute
- Press *Done*

Add Module (7)

- (Step 2 of 3)
- May add more parameters
- Repeat until all parameters are added
- May add more later
  - using Object Editor
  - editing V directly
  - drag and drop from Standard Objects library
- When done, press *Next >*

Add Module (8)


Module with all of the child objects defined in this tool
Add Module (Step 3 of 3)
- Specify process into which this module is linked
- Specify file for module's source code (relative to indicated directory)
- Set additional (advanced) properties, if needed
- Use buttons to generate, edit, and/or compile code as desired
- Click on "Done" when finished
SampleModule
update = "my_update"
in value = ?
Process in which object resides
Inherit (express)
Directory for source: /usr/local/express/express/mydir
Source filename
mycode.c
Advanced properties
build_dir
mydir
The name of a directory to store build files
Edit Source
Compile Process
Use Project->Save As... to create a writable project
< Back
Done
Cancel
Help
Module Process (Inherited)
Current Path for Module Source
Source Filename
Properties and values

Add Module (9)

- (Step 3 of 3)
- Module must execute in a particular process
  - default is to *inherit* from parent
  - this will be a library, on which attributes may also be set
- The *express* process is where majority of modules execute
  - fast and efficient

Add Module (10)

- (Step 3 of 3)
- *user* process used for quickly adding modules to running environment
  - won’t kill the entire express application
  - communicates with other express modules via shared memory
  - slow and inefficient
  - however to add modules to the *express* process requires exiting, recompiling, and then restarting the application
- From Express 3.3 dynamic loading of modules into the *express* process is possible by placing them in a shared library

Add Module (11)

- (Step 3 of 3)
- Directory for source is displayed
  - Must have created a project, which allows V code to be written
  - Express without a user defined project is read-only
- Source code containing the function must be written to a file
  - Specify the source filename
  - Express will generate skeleton code, which user can edit

Advanced Properties

- A number of properties may be selected from the menu
  - *build_dir*
    directory from root directory of project where files will be generated
  - *out_src_file*
    express generated code can be placed into a named file
  - out_hdr_file
    express generated header information can be placed into a named file
- Other properties include:
  - *cxx_src_files*, *cxx_hdr_files*, *c_src_files*, *c_hdr_files*, *f_src_files*, *link_files*, *init_code*, *pre_init_code*, *hdr_code*
- Common properties only given here

Add Module (12)

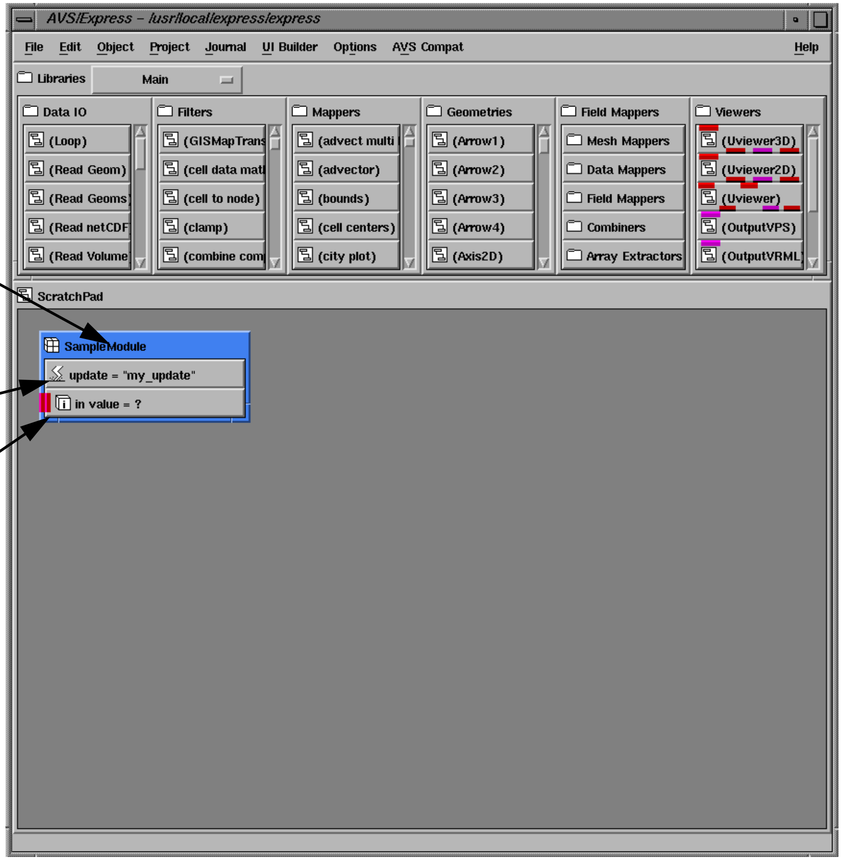
- (Step 3 of 3)
- Source file may be edited from here
  - User writable project must have been saved
- Process may be compiled from here
  - source code will be generated if it doesn't exist already
- The Express process cannot be recompiled without exiting
- However all code will be generated along with *makefiles*
- Once exited, *express* may be compiled from the command line
- When complete, press *Done*

New Module

Module in the Scratch Pad

Module Method

Parameters



NB. Move object to the workspace libraries so that it may be instanced and executed

New Module (2)

- Back in the Scratch Pad, the new module can be seen
  - Open the module to reveal the methods and parameters as objects
- Object may be dragged to one of the *Workspace* libraries
  - *Save Project* from *Project* menu
- Select module and then *compile* from *Project* menu
  - Express will ask if skeleton code should be generated (first time only)
  - Express will not change any existing files that contain user source code
  - Generated files as specified in advanced properties will change
  - Generated files are different on different platforms

Project Structure

- Files and directories that Express uses:
- Project root directory:
  - *avsenv*, contains path information
  - *v/* contains the files *templ.v*, *proc.v*, and later the files *lib_(v)xp.v*
  - *bin/* contains executables for express and user process
  - *lib/*, *include/* and *motif_ui/* contain express generated files
- User specified *build_dir* property, may result in further directories in the top-level

Project Structure (2)

- The file *templ.v* generally contains all V definitions
  - modification of the Templates library and hence all objects
  - sensitive file which is important to Express
  - it is possible to save definitions in other files
- Saving Projects and Objects
  - project->Save Project
  - object->Save Objects having selected which objects to save

Makefiles

- express.mk or user.mk
- use *base -comp_proc express* command
- PC users may compile from command line too
  - requires Microsoft Visual C++
  - compile from MS-DOS prompt
  - nmake -f express.mk

Task 1

Refer to Exercise 3:1 in the Workbook.

- ❑ Use the Add Module Tool
- ❑ Create a simple module to add two numbers together
- ❑ Generate the skeleton code
- ❑ Add the code provided to implement the method

Network Editor

- Drag and Drop components
  - possible to create module from parts available in libraries
  - the *Standard Objects* library
  - scratch pad used as building area
- Module Components
  - module types
  - method types
  - primitive and high-level data objects

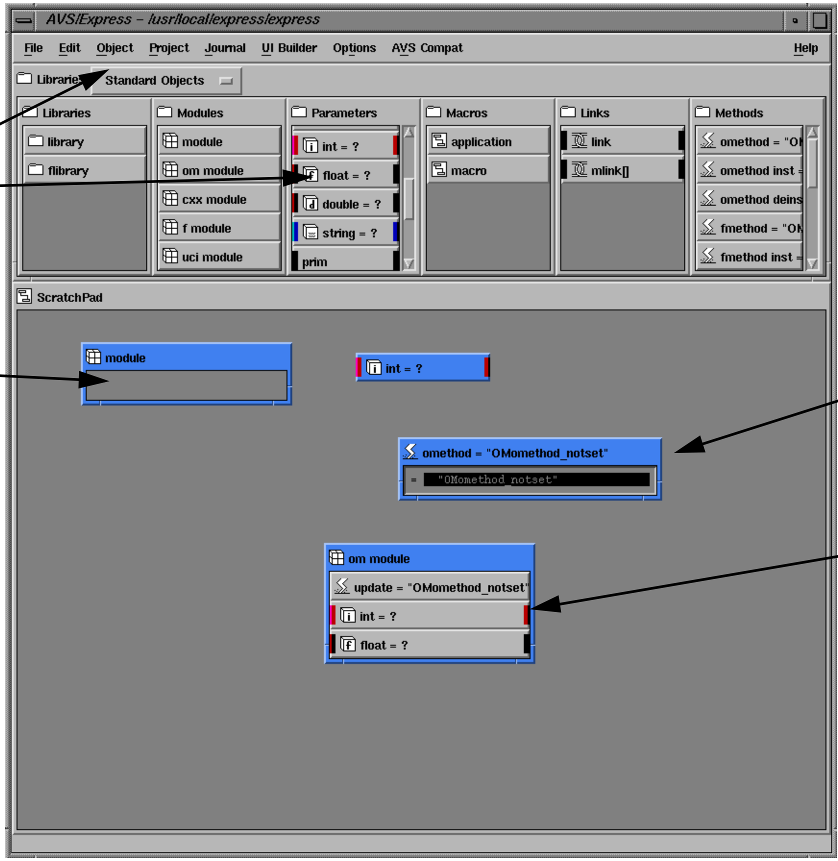
Network Editor (2)

Standard Objects Library and the available objects

Empty Module

A Method

Module with components from libraries



Task 2

Refer to Exercise 3:2 in the Workbook

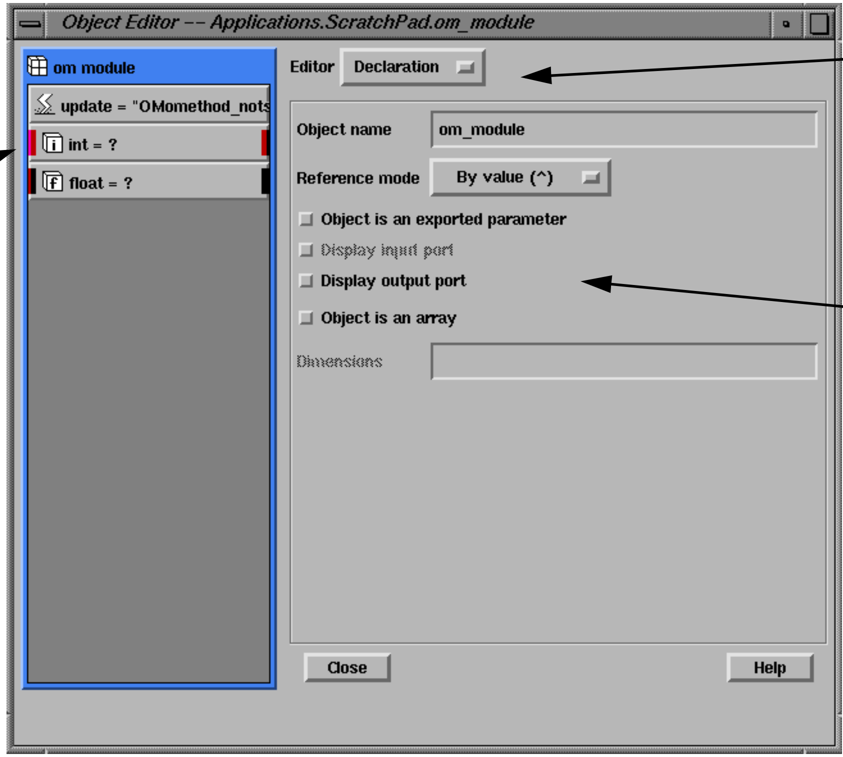
- Using the Scratch Pad application
- Drag and Drop components from the libraries to create the same module as Task 1
- Specify the same method/function name

Object Editor

- ❑ Similar to the Add Module Tool
  - ■ may now select which page to view
- ❑ Used for editing existing modules
- ❑ May add, modify or delete components
- ❑ May change attributes and some properties

Object Editor (2)

Module with the child objects



Select Editor Page

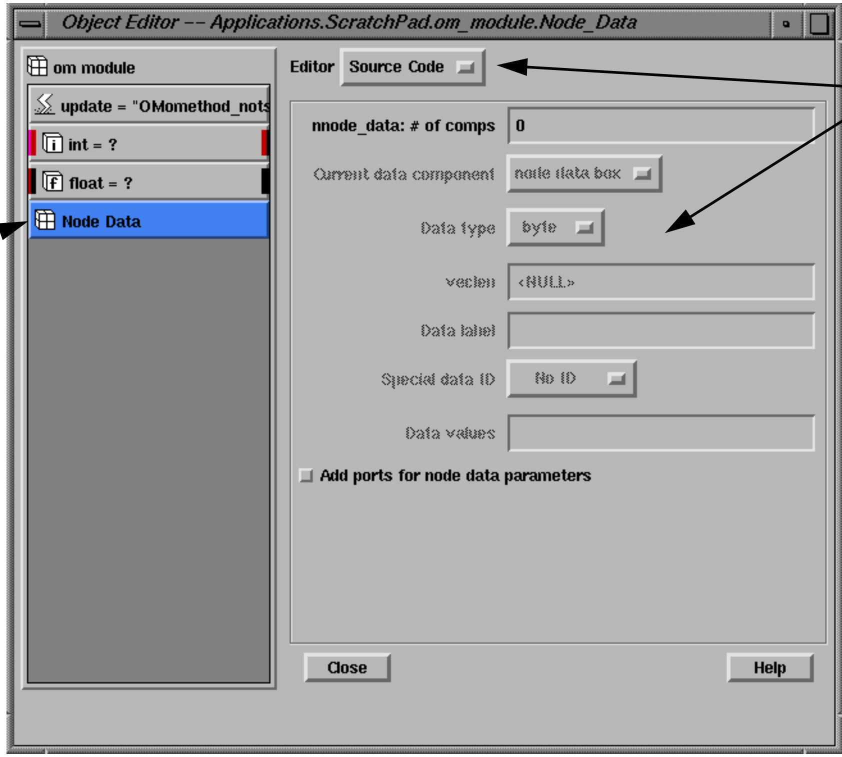
Same options as Add Module Tool

Advanced Editor Pages

- High-level datatypes, e.g., Field, Node_Data have advanced editor pages of their own
  - available in both Add Module Tool and the Object Editor
  - editors which change accordingly with the object selected
- Allows specification of some or all parameters found in high-level data objects
- More about manipulation of Field objects later

Advanced Editor Pages (2)

High-level data object (Express built-in type)

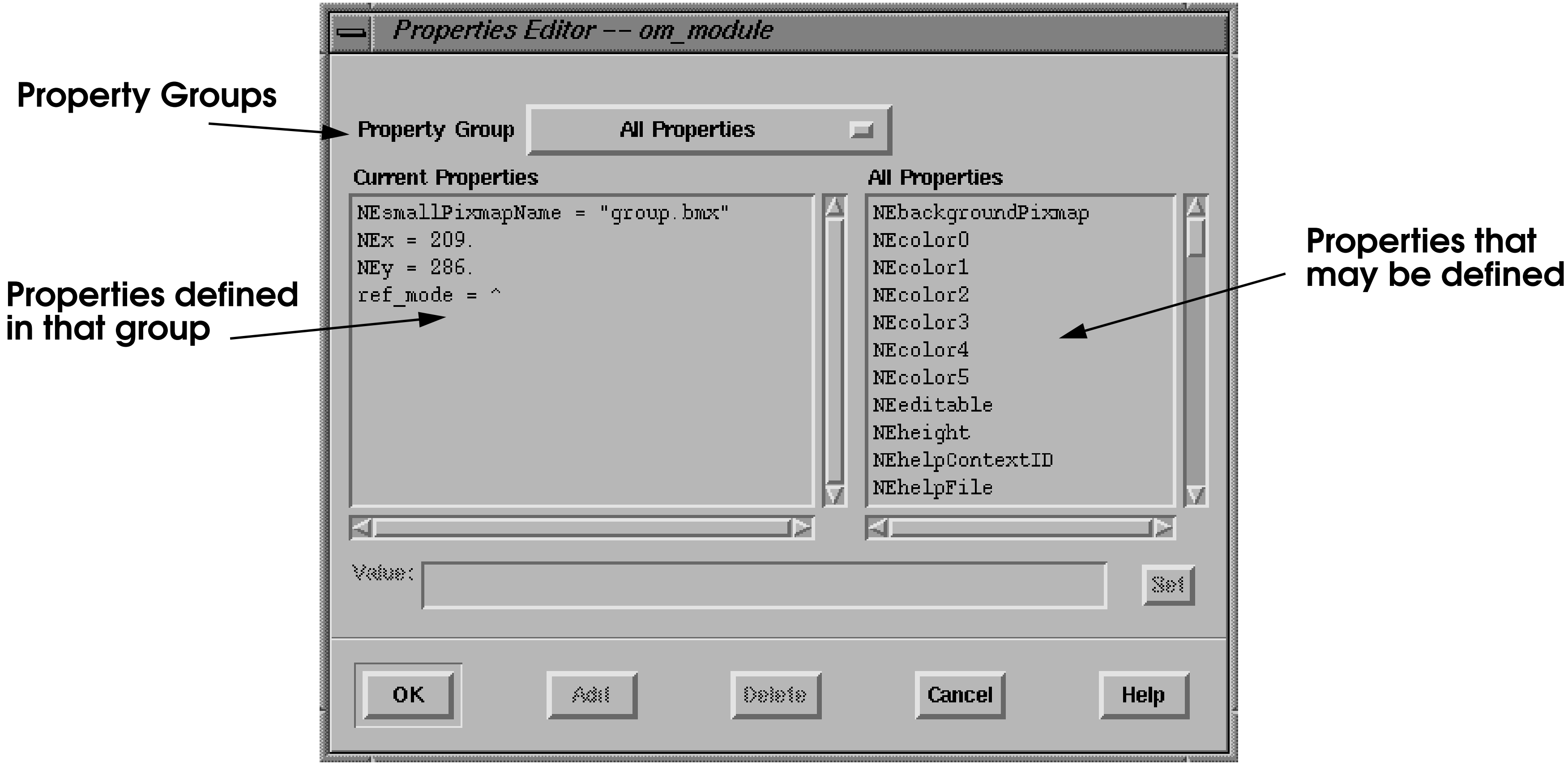


Source Code Page for Node_Data type

Properties Editor

- ❑ Access to all properties that each object has
- ❑ Includes those found in editors, with many more besides
- ❑ Available in pages, or all at once
- ❑ Network Editor properties found here
  - specify port colours
  - position of objects within macros

Properties Editor (2)



Task 3

Refer to Exercise 3:3 in the Workbook

- ❑ Create a group to define the data type Point2D
- ❑ Create a module to calculate the distance between two points
- ❑ Use Property Editor to customise the port colours

Part III
V Definition Language

As used by Express

- ❑ Define all objects found in libraries
- ❑ Redefine any object
- ❑ Modify exisiting objects, changing the functionality
- ❑ Add or delete objects
- ❑ Instance objects for use by other objects
- ❑ Connect objects together
- ❑ Name objects
- ❑ Hierarchically organize the objects
- ❑ Saving applications

Overview of V

- Definition language
  - not for programming
  - no control flow or conditional statements
  - basic arithmetic and mathematic support
  - basic array support
- Text and Binary formats
  - text is readable and editable
  - compiled at run-time by Express
  - may be compiled permanently to improve speed and efficiency

Overview of V (2)

- Why use V?
  - writing V code is more powerful
  - more efficient once effort has been made to learn it
  - interactive manipulation of objects via VCP
- Features
  - comments
  - statements
  - functions
  - commands

Statements

- Add or replace a new object
  - added at current scope
  - specify type based on existing type (base type or template)
- Modify an existing object
  - change its attributes, properties and value/reference
  - open and modify its subobjects
- Delete an object
  - remove the object and its subobjects

Add Statement

- Uses the following clauses:
  - object_type[+object_types+object_attributes]
  - [reference_mode]
  - object_name
  - [<properties>]
  - [array_declaration]
  - [{subobject_statements}]
  - [value_expression];
- [] indicate optional clauses, semi-colon to terminate statement

Add Statement (2)

- object_type[+object_types+object_attributes]
  - list is delimited by the '+' symbol

1. *Use previously defined object*

```
int x;
Point topleft;
```

2. *Use type with boolean attributes*

```
int+write distance;
```

3. *Use list of (high-level) types and boolean attributes*

```
Mesh+Node_Data+read+notify+req out;
```

- Note that the list of types must be group objects
- Primitives cannot be merged in this way

Add Statement (3)

- [reference_mode]

1. *^ (by-value)*
   - default mode: object contains its own value

   ```
   group ^x; //same as group x
   ```
2. *& (by-reference)*
   - references the subobjects of another object

   ```
   group &x;
   ```
3. ** (by-pointer)*
   - pointer to object; must be explicitly dereferenced to access subobjects

   ```
   group *x;
   ```

Add Statement (4)

- object_name

1. *Specify a new name*
   - unique name amongst the object's siblings
   - consist of letters numbers, # and _ (which appears as a space in NE)
   - must not start with a digit
2. *Specify an existing name*
   - name one of the existing siblings
   - object is deleted and replaced with new one

Add Statement (5)

- [<properties>]
- Object's properties are:
  - inherited from the parent
  - or defined (re-defined) here
- List enclosed in angle brackets <> and delimited by commas.
  - set the input and output port levels to 2.

```
int thing <NEportLevels=2>;
```

- Array of values uses curly braces
  - set input port level to 2, and the output port level to 0.

```
int thing <NEportLevels={2,0}>;
```

Add Statement (6)

- [array_declaration]
- A C-style array declaration for arrays

1. *One dimensional array*
   - integer array with 10 elements

   ```
   int thing[10];
   ```

   - integer array with number of elements set in sibling object num

   ```
   int thing[num];
   ```

Add Statement (7)

2. *Unspecified bounds*

   - treate as an array where dimension will be set later

```
int thing[];
```

   - reference mode to allow connection to array of any length

```
int &thing[];
```

   - only allow connection to array of specified length

```
int &thing[10];
```

3. *Multidimensional array*

   - can use explicit values or other objects

```
int thing[x][y];
```

Add Statement (8)

- [{subobject_statements}]
- Contains one or more subobjects
  - macros, groups (modules) and libraries, but not primitives
  - except for min, max or value constraints
- Subobjects are inherited from template, or defined here
- Curly braces can contain any number of Add, Modify or Delete statements
  - subobjects themselves can be opened in this way
  - hence the hierarchical nature of the objects

Add Statement (9)

- [value_expression]
- Scalar or array value expression

1. *Assignment (=)*
   - assign a constant to the object

```
int thing = 4;
int thing[3] = {4,9,16};
```

Add Statement (10)

2. *Connection (=>)*

- connection to another object

```
int thing => x;
```

- connection to an expression

```
int thing => 3*(x+y);
```

- connection to an array of objects

```
int thing[] => {a,b,c};
```

Modify Statement

- Uses the following clauses:
  - [reference_mode]
  - object_name
  - [additional_types_and_attributes]
  - [delete_attributes]
  - [properties]
  - [array_declaration]
  - [subobject_statements]
  - [value_expression];

Modify Statement (2)

- Change the object's reference mode

```
&thing;
```

- Add more types or attributes

```
thing+read;
```

- Delete attributes (but not types)

```
thing-read;
```

- Add or modify properties

```
thing<NEportLevels={3,0}>;
```

Modify Statement (3)

- ❑ A new array declaration for the object

```
thing[15];
thing[x][15];
```

- ❑ Changes to the sub-objects (Add, Modify and Delete)

```
thing {
 x = 100;
};
```

- ❑ New assignment

```
thing = 22;
thing => fred;
```

  - ■ to break a connection and set to NULL

```
thing =>;
```

Delete Statement

- Uses the following clause:
  - -object_name;
- Delete the object by prepending the '-' symbol;

```
-thing;
```

- Delete subobject by opening with curly braces

```
thing {
 -x;
};
```

Functions

- Built-in functions for value expressions
- A number of categories available
  - mathematical
  - logical
  - array
  - string
- Example

```
result => sqrt(in);
```

Commands

- Built-in commands of V and the VCP
  - prefixed by the $ symbol
  - don’t require ‘;’ postfix
- Perform special operations
  - access objects
  - create or delete objects
- Examples
  - $get_array, $print, $destroy
- Useful in the VCP

VCP

- The V Command-line Processor
  - interactive command line for Express
  - use V to access and modify objects
  - navigation of existing applications and objects
  - has a current open object
- Use V functions to interact with objects
- Use V statements to create, modify and delete

VCP (2)

- Opening Sub-Objects
  - open, examine contents and close with:

```
name {
 $list
};
```

- Applications

```
DefaultApplication {
```

  - assign values to objects and see changes in Network Editor
  - see V code for current object

```
$print
```

Object Type

- Specify the template to be used to create a new object
  - a primitive type, from int, float, double, string etc.
  - a pre-defined high-level type, e.g., Node_Data, Grid etc.
- May also use existing objects as templates too

```
int myint;
```

  - will not have a value set by default

```
myint = 5;
```

  - will set the value

Object Type (2)

- Use object as a template

```
myint other;
```

  - will use current *myint* as template

```
$int other
```

  - will return the value of 5

- Without a type in the statement, an existing object is modified

```
other = 4;
```

Attributes

- For use with methods
  - the *read*, *req*, *write*, and *notify* attributes
- Added after the object type,

```
int+read+notify+req name
```

- Modules have attributes too

```
module+notify_inst
```

- Express includes common definitions with macro names

```
int+Iparam name;
```

  - is equivalent to

```
int+read+req name<NEportLevels={2,0}>;
```

  - note that this also includes the property <NEportLevels={2,0}>

Object Name

- Readable name used for objects
  - not too long or it makes editing V cumbersome and error-prone
  - names should be unique
  - drag and droppping objects with same name will result in Express appending a unique #n suffix to the object name.
  - drag and dropped objects take type as default name, e.g., int and int#1

Reference Modes

- Three modes for an object to take:
- by-value (^)
  - real instance of the object
  - the value is stored here, although it may be linked to another
- by-reference (&)
  - reference to a real instance of the object of same type
  - values accessed here, but are really references to another real instance

Reference Modes (2)

- by-pointer (*)
  - points to a real instance somewhere in the process memory
  - advanced use of objects
  - pointer and target must reside in same process
  - pointer edited to reference many objects without direct NE connections

Properties

- ❑ Optionally specified after object name
- ❑ Delimited by angle brackets <...>
  - network editor properties appear here
  - NEx, NEy, NEvisible, NEcolor
  - NEportLevels
  - source code filename and process specified here
  - build_dir, src_file, and other properties

```
module+notify_inst mymod<process=”user”,src_file=”mymod.c”>;
```

Matching Objects

- Reference (&) and Pointer (*) objects must reference an object of an acceptable type
  - primitive data types are automatically cast to the correct type

```
int x;
float y => x;
```

- Referenced object must at least contain the child sub-objects that match the referencing object template
  - this includes the name of subobjects as well as the type
- However, the referenced object may have *more* child objects
  - the referencing object will also have these additional objects whilst connected

Matching Objects (2)

- Given the following groups:

```
group one {
 int a;
};

group two {
 int a;
 int b;
};
```

  - instance of *one* can connect to instance of *two*:

```
one &myone => two;
```

  - as *two* does contain required sub-object '*int a*'
  - instance of *two* cannot connect to instance of *one*:

```
two &mytwo => one; // will not work
```

  - as *mytwo* requires at least the two sub-objects '*int a'* and '*int b'*

Matching Objects (3)

- Matching requirements can also extend to specific values

```
group one {
 int a;
};

group two {
 int a = 4;
};
```

  - instance of *one* can connect to *two*

```
one &myone => two;
```

  - as it does not require a specific value
  - instance of *two* cannot connect to *one*

```
two &mytwo => one; // does not work
```

  - as *mytwo* requires that referenced object has an ‘*int a*’ which is set to 4

Multiple Inheritance

- When creating new objects, it is possible to combine two or more types to form a new type
- In V, the types are combined with the *+* operator

```
Grid+Node_Data myobj;
```

  - In the NE hold down CTRL and drop each additional type

- If the merged types have child objects with the same name, the type added last in the list takes precedence

Other Objects

- Link Objects
  - simply link objects to each other, avoiding direct connection
  - allows sensible and known name to be used
  - external connections made by name
  - not possible to know name of object outside current scope
  - no idea what is going to use this object
  - connects scalar objects
- Mlink Objects
  - connects array objects
  - allow multiple inputs to one port by creating an array of them

Task 4

Refer to Exercise 4:1 in the Workbook

- ❑ Use the VCP to create primative objects
- ❑ Use V functions to examine the objects

Task 5

Refer to Exercise 4:2 in the Workbook

- Use the VCP to create group objects
- Use V functions to examine the objects

Task 6

Refer to Exercise 4:3 in the Workbook

- ❑ Use the VCP to create array objects
- ❑ Use V functions to examine the objects
- ❑ Instance the modules from the previous Tasks
- ❑ Use the VCP to examine and modify the objects

Task 7

Refer to Exercise 4:4 in the Workbook

- Create a simple V Code module that uses the loop module to generate a series of filenames that could be used to read a set of files
- Shows use of inbuilt functions

Task 8

Refer to Exercise 4:5 in the Workbook

- Create a V Code module that will sum together the first four values of two arrays
- Shows use of sub-arrays and inbuilt functions

PART IV
C and C++ Source Code

C API

- Object Manager C Access Library
  - object ID
  - object Name
- Status and Error Messages
  - return status
  - using the error function
  - useful for debugging

C API (2)

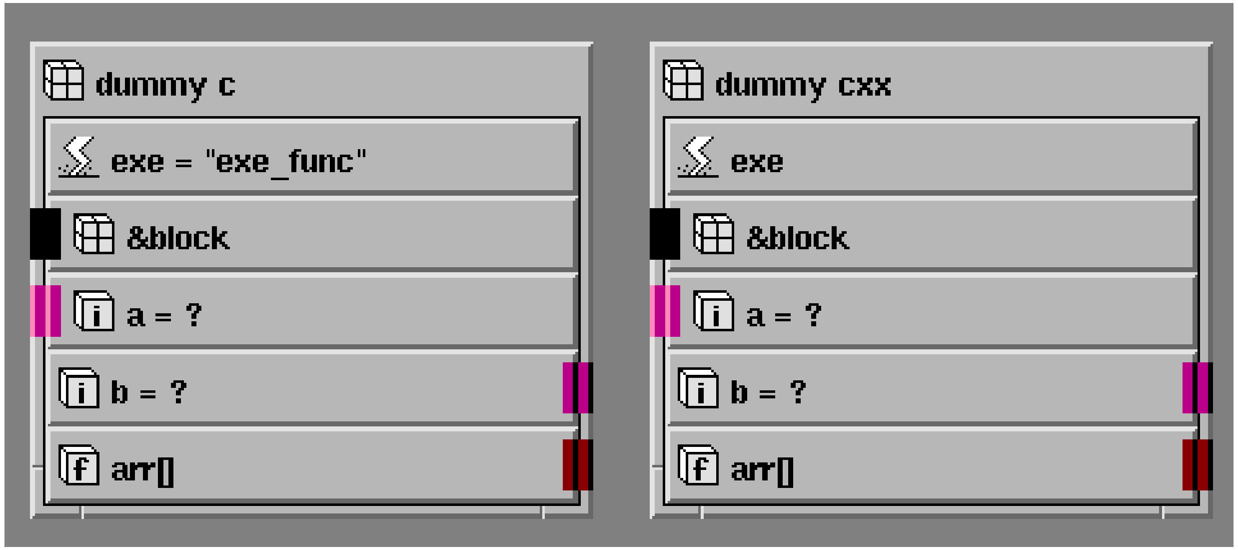
- Accessing Primitives
  - standard C types
  - standard OM calls
  - read and write access
- Accessing Arrays
  - standard C types
  - standard OM calls
  - supporting macros and functions
  - memory allocation and deallocation

C++ API

- Express generates C++ class representation of Express Object
- Access member objects as
  - primitive data types
  - high-level data types (e.g., fields) with classes
- Why use C++?
  - much easier access of data members, no string names
  - all OM and FLD C functions still available
  - enforces object-oriented approach

C and C++ API Example

- identical dummy modules one using an OM method the other using an OMX method



C API Example Code

```
#include "/home/cluster/zzcgujp/projects/modwrite/./user.h"
int exe_func(OMobj_id dummy_c_id, OMevent_mask event_mask, int seq_num)
{
 OMobj_id block_id;
 int a;
 int b;
 int c;
 float *arr = NULL;

 /*  Get input values   */
 if (OMget_name_int_val(dummy_c_id, OMstr_to_name("a"), &a) != 1) a = 0;
 /*  set number of elements in array: */
 b = a/2;
 OMset_name_int_val(dummy_c_id, OMstr_to_name("b"), &b)

 /* access member of group */
 OMget_obj_id(dummy_c_id, OMstr_to_name("block"), OM_OBJ_RD);
 if (OMget_name_int_val(block_id, OMstr_to_name("c"), &c) != 1) a = 0;
```

C API Example Code (2)

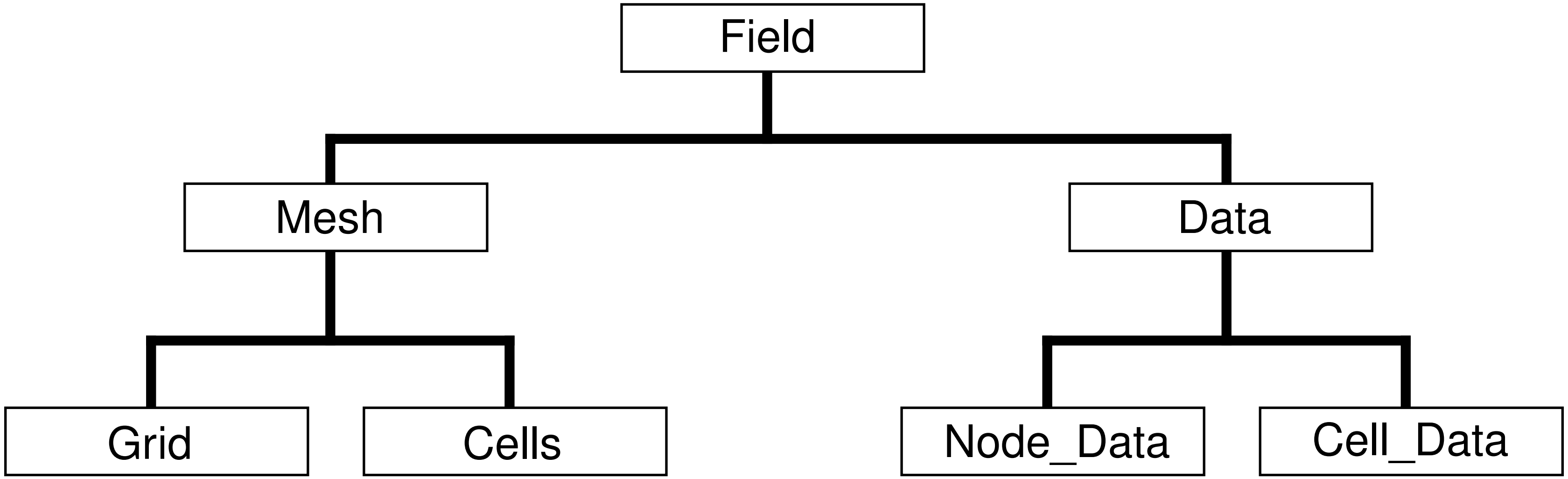
```
 /*  allocate array arr */
 arr = (float *)ARRalloc(NULL, DTYPE_FLOAT, arr_size, NULL);

 /*  fill in array arr */
 OMset_name_array(dummy_c_id, OMstr_to_name("arr"),
  DTYPE_FLOAT, (void *)arr, arr_size, OM_SET_ARRAY_FREE);

 /* alternatively, if the dimensions of arr array are set at
 this point you can use this call that allocates memory */
 arr = (float *)OMret_name_array_ptr(dummy_c_id,
  OMstr_to_name("arr"), OM_GET_ARRAY_RW, &arr_size, NULL);

 return(1);
}
```

C++ API Example Code

```
#include "/home/cluster/zzcgujp/projects/modwrite/./user.h"
int WORKSPACE_1_dummy_cxx::exe(OMevent_mask event_mask, int seq_num)
{
 // block (group read req notify)
 // a (OMXint read req notify)
 // arr (OMXfloat_array write) dimensions [b]
 int arr_size;
 int c;
 float *arr_arr;

 // access primitive members and set size of array
 b=a/2;
 // access group members
 c=block.c;

 // access array
 arr_arr = (float *)arr.ret_array_ptr(OM_GET_ARRAY_WR);
 if (arr_arr) {
  ARRfree((char *)arr_arr);
 }
 return(1);
}
```

Field Overview

- Field hierarchy comprises:



Field Hierarchy

- Mesh
  - geometric description of where and how the data is located and organized in space

1. *Grid*
   - the location of the nodes in space, coordinates
2. *Cells*
   - the node connectivity
   - specifies types of cell, e.g., tri, polygon, hex
   - informatation for each type forms a cell_set

Field Hierarchy (2)

- Data

1. *Node_Data*
   - numeric data associated with each node
   - any number of components which may be scalar or vector
2. *Cell_Data*
   - numeric data associated with each cell
   - any number of components which may be scalar or vector

Field Types

- The Field object type is available in a number of forms
  - high-level Field types for uniform, rectilinear, spherical and cylinder
  - low-level components for just specifying the grid, cells, node_data etc.
- High-level types are combinations of low-level components
  - Field_Unif is built with:

```
Header+Mesh_Unif+Data_Struct Field_Unif;
```

  - and similarly, Mesh_Unif is built from:

```
Grid_Unif+Cells_Struct+Xform Mesh_Unif;
```

  - which in turn are built from smaller components

Field Manipulation

- Accessing AVS/Express Field objects
  - supporting functions and classes
- Skeleton code is produced with good examples of use
  - the ndim and nspace values
  - set local data
  - call functions

```
FLDset_ndim
FLDset_nnodes
FLDset_node_data_ncomp
FLDget_node_data_label
...
```

Field Manipulation (2)

- Mesh Type
  - the V definition affects what information is required
  - using high-level components automatically ‘fills’ some of the requirements
  - set mesh type
- Adding Grid Data
  - fill coordinate array locally
  - use function to apply to field

Field Manipulation (3)

- Adding Node Data
  - specify number of components that node data has
  - specify the vector length for each components
  - specify the primitive or high-level type for each component
  - fill node data arrays locally
  - use function to apply component to field output
- Adding Cells and Cell Sets
  - cell sets define a number of cells
  - each cell defines connectivity between nodes of data

Field Manipulation (4)

- Geometry Cell Sets
  - geometry types are defined as cell sets with predefined names
  - specify type of cell set; polyline, triangles, polyhedra etc.
  - use functions to create cell sets; e.g., FLDset_cell_set
  - specify the number of connections needed
  - support functions will determine size of array needed for geometry type

Field Manipulation (5)

- Geometry Types
  - lines - each line requires two nodes, connectivity array is 2 x num_lines
  - polylines - set of lines joining consectutively indexed nodes , array is 2 x num_lines
  - more effort required for efficient types, but performance is boosted
- Adding Cell Data
  - use functions to set cell_set type
  - fill cell connectivity array

Task 9

Refer to Exercise 5:1 in the Workbook

- ❑ Implement a JuliaSet module which will create Node_Data
- ❑ Output a Uniform Field that will contain 2D data
- ❑ Place into a macro

Task 10

Refer to Exercise 5:2 in the Workbook

- ❑ Implement a Cube module to generate geometrical data
- ❑ Create module to output a mesh, and use skeleton code
- ❑ Fill in blanks in code to output nodes and cells to form a cube
- ❑ Instance module and connect to viewer
- ❑ Extend module so user can specify scalings for each axis

PART V
User Interface

UI Kit

- AVS/Express comes complete with a UI kit library
- Allows widgets to be connected to module/macro parameters
- Each object provides either
  - low-level widget type, e.g., slider, dial, or typein
  - high-level UI objects, e.g., file browser
- Combine UI objects to form UI Macros
  - use group object to combine the parameters of the module
  - more later about organization of the objects in a project

Parent

- Each UI object must have a parent
  - a frame in which the widget will reside
  - top-most parent will be a window
- AVS/Express applications provide such frames (panels)
  - SingleWindowApp has Module menu item
  - each macro's user interface appears as a separate panel, accessed via the module selection menu

Panel

- UImod_panel object is used to:
  - provide a frame for all child widgets to connect to and be placed inside
  - and to connect to the application module panel if provided
- Each child UI object takes an input to connect to UImod_panel

UI Parameters

- UI objects are group objects with many sub-objects inside
  - sub-objects control appearance and functionality
- Also contain the parameters
  - set the current value
  - set range for the widget if applicable
- Parameters are standard primitive types
  - hence they can be imported
  - can be made to reference another object
  - connect to common parameter block

Example

- Attach a slider to an integer value
- Create a UImod_panel
- And a slider widget

Task 11

Refer to Exercise 6:1 of the Workbook

- ❑ Add a User Interface to the Threshold module/macro
- ❑ Export the module parameters, and import into UI objects
- ❑ Create a SingleWindowApp and instance module and UI
- ❑ Use the sample application and use the UI to manipulate the data

PART VI
Project Organization

Storage of V

- Avoid templ.v to protect projects
  - one error by user or Express can damage entire project
  - very hard to locate code for an object
  - default layout is difficult to follow
- Separate Files
  - possible to save objects or groups of objects in their own V file
  - damage limited probably to one file
  - files are smaller and easier to edit

Using Libraries

- Separate files enforces hierarchical organization of objects and files
- Avoid old syntax - $include
  - will not be supported in future releases for code management
- Use the *flibrary* command to add libraries containing new objects in own files
- e.g., “iac_proj/thresh/thresh.v” Threshold;
  - creates library in parent (another library) which contains objects defined in the file thresh.v
  - those files in turn may create more libraries in the hierarchy

Using Libraries (2)

- The *templ.v* file
  - now only contains lines to load libraries
  - should still avoid using *Save Project* directly, as this will append other V to the templ.v file
  - it will also change the format and layout of existing V code

Properties

- Useful properties that avoid common problems
- Recommended for IAC project submissions, but also for general project development

build_dir

- Use this to define a directory where source code can be found
- Default is top-level of the project directory
- Can get messy with lots of objects
- The IAC uses *iac_proj/<projname>/* as the directory
- Note that IAC assumes individual submissions as projects rather than objects
- *iac_proj/* used as this does not conflict with a standard Express directory

build_dir (2)

- Avoid names that conflict with Express installation, e.g., *src*, *modules* etc.
- These act as overrides to the original installation
- Any part of Express can be changed!

process

- Used to specify which process the object will execute in
- Default is to inherit from parent (typically *express*)
- Recommended that this is not used (for submissions)
- *user* process used for quick development
- compiling *express* process is slow

out_hdr_file & out_src_file

- Generated source code requires Express headers and other code
- Normally this is generated within the source code
- These two properties force this information to be written to different files
- E.g., gen.h and gen.cxx (within a unique build_dir)
- Header information includes specification of process
  - e.g., #include “user.h”
- If process is changed, the source code would have to be edited by hand

out_hdr_file & out_src_file (2)

- Express will not edit source code
- With these properties, the header information can be rewritten every time something changes without affecting the code
- Also makes code more portable
- IAC submissions do not include these files

Absolute Pathnames

- Should be avoided at all costs
- Usually not portable between platforms or even users
- This applies to all other properties that require paths
- Example applications typically load a sample dataset
  - e.g., /usr/express/data/field/hydrogen.fld
  - will not work on a PC, which uses C:\Express\data\field\hydrogen.fld for example
- Should use environment variables in this instance
  - e.g., $XP_ROOT/data/field/hydrogen.fld

Absolute Pathnames (2)

- PC Express will recognise the UNIX / separator
- May require manual editing of the V file

IAC Project Files

- Overview
  - the IAC submissions and projects available from the repository use the above rules
  - makes it easier for users to download and add if they are consistent in style
  - makes it easier for the IAC to manage and check portability

IAC Project Files (2)

- Simplifying Structure
  - a project is typically not just a module
  - group objects are used to specify parameter blocks
  - user interface macros are needed
  - other abstractions make use of a module much easier
  - e.g., functional macro, which processes fields, leads to a library of likewise objects

IAC Project Files (3)

- Standard Files
  - avoids editing templ.v file
  - references iac_proj/iac_proj.v file instead
  - this includes lines to load the various downloaded IAC projects
  - each project goes into a directory of its own in iac_proj/
  - each project is placed into a library of its own
  - this is within the Templates.IAC_PROJ library.
  - the IAC project files also create a new library page called IAC
  - viz edition users cannot access Template.IAC_PROJ directory

PART VII
IAC Threshold Project

Requirements

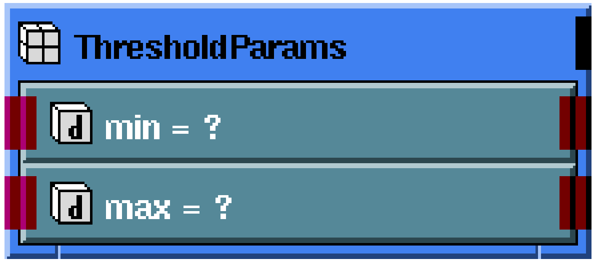
- Module and supporting macros to implement a simple Threshold filter
  - process only Node_Data, as Mesh is unimportant
  - identify the parameters: min and max threshold values
  - place parameters into a common parameter block group
  - produce high-level Functional Macro to allow processing of Field
  - produce high-level UI Macro to interface to parameter block
  - produce high-level User Macro to encapsulate the whole functionality

Parameter Block Group

- ❑ Require only min and max threshold values
  - ■ use double precision float to specify range for any type of data
- ❑ Group object composed of:

```
group ThresholdParams <NEportLevels={0,1}> {
 double min <NEportLevels={2,2}>;
 double max <NEportLevels={2,2}>;
};
```

- ❑ And appears as:



Low-Level Module

- Needs only to process Node_Data
  - this forms both the input and output type
  - always specify the minimum amount of data required by the module
- Module also requires parameter input
  - achieved via the common parameter block ThresholdParam
  - also common to repeat the parameters as internal objects to the module and then connect to the parameter block

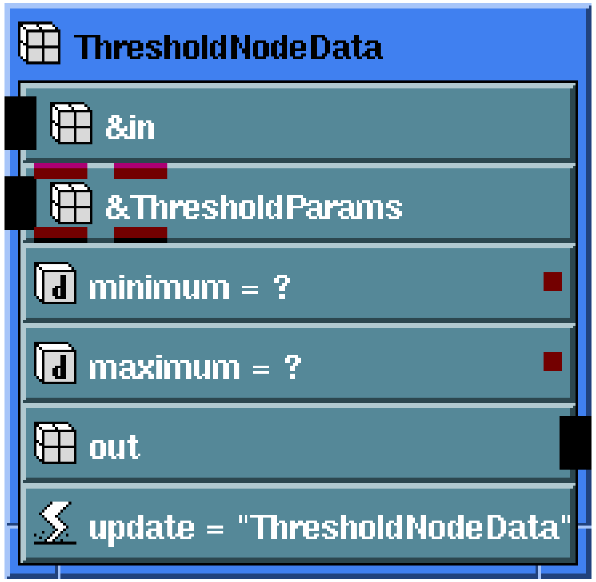
Low-level Module (2)

- Module composed of:

```
module ThresholdNodeData <src_file=”thresh.c”> {
 Node_Data &in <NEportLevels={2,0}>;
 ThresholdParams &ThresholdParams <NEportLevels={2,0}>;
 double minimum => ThresholdParams.min;
 double maximum => ThresholdParams.max;
 Node_Data out <NEportLevels={0,2}> {
  nnode_data+nres => <-.in.nnode_data;
  nnodes+nres => <-.in.nnodes;
 };

 omethod+notify_inst+req update (
  in+read+notify+req,
  minimum+read+notify+req,
  maximum+read+notify+req,
  out+write )
 = “ThresholdNodeData”;
};
```

Low-level Module (3)

❑ And appears as:



Functional Macro

- Macro required to process Field data type
  - rather than just Node_Data
  - for use in general applications
- Must input and output Field
  - new Node_Data from ThresholdNodeData module
  - must merge this with original Mesh from input field

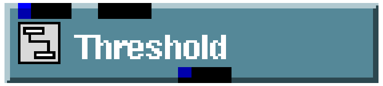
Functional Macro (2)

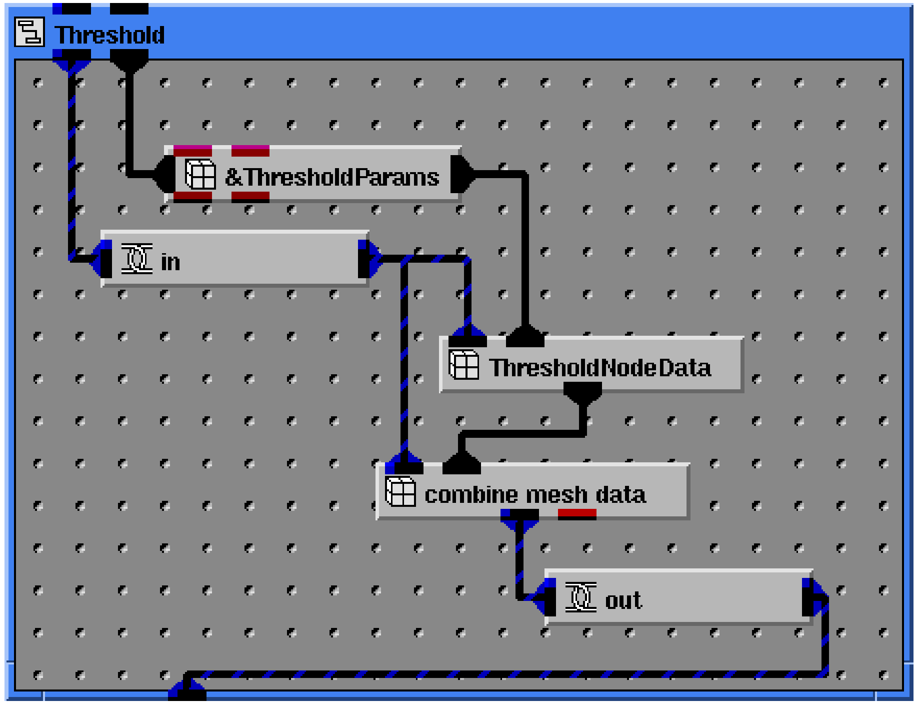
- Macro composed of:

```
macro Threshold {
 link in <NEportLevels={2,1},NEx=100.,NEy=50.>;

 IAC_PROJ.Threshold.ThresholdMods.ThresholdParams
  &ThresholdParams <NEportLevels={2,1},NEx=300.,NEy=50.>;

 IAC_PROJ.Threshold.ThresholdMods.ThresholdNodeData
 ThresholdNodeData <NEx=100.,NEy=250.> {
  in => <-.in;
  ThresholdParams => <-.ThresholdParams;
 };

 FLD_MAP.combine_mesh_data combine_mesh_data <NEx=200.,NEy=342.> {
  in_mesh => <-.in;
  in_nd => <-.ThresholdNodeData.out;
 };

 link out <NEportLevels={1,2},NEx=100.,NEy=420.>
  => combine_mesh_data.out;
};
```

Functional Macro (3)

- And appears as:





UI Macro

- User Interface macro required to manipulate parameters
  - widgets attach to min and max threshold values
- UI Macro composed of:

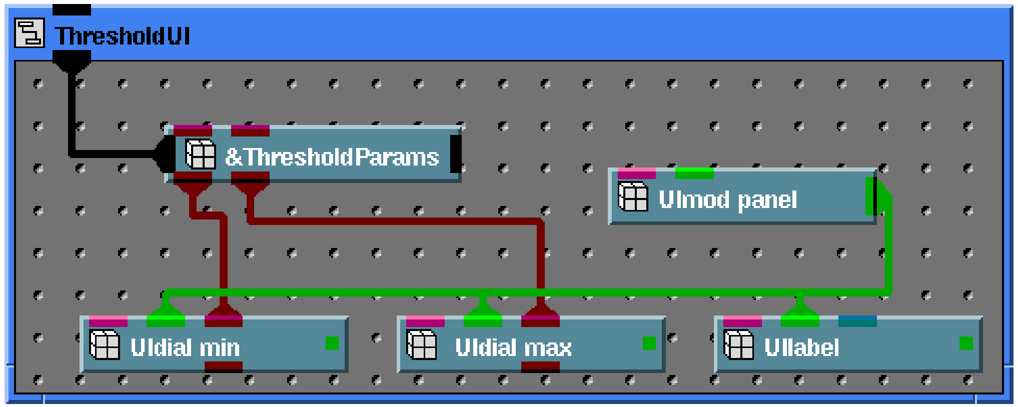
```
macro ThresholdUI {
 IAC_PROJ.Threshold.ThresholdMods.ThresholdParams
  &ThresholdParams <NEportLevels={2,1},NEx=187.,NEy=120.>;

 UI.UImod_panel UImod_panel <NEx=330.,NEy=20.> {
  title = “Threshold”;
 };

 UIdial UIdial_min <NEx=440.,NEy=120.> {
  parent => <-.UImod_panel;
  value => <-.ThresholdParams.min;
  title = “Minimum”;
  y = 50;
  width = 100;
 };
```

UI Macro (2)

```
 UIdial UIdial_max <NEx=440.,NEy=170.> {
  parent => <-.UImod_panel;
  value => <-.ThresholdParams.max;
  title = “Maximum”;
  y = 50;
  x = 120;
  width = 100;
 };

 Controls.UIlabel UIlabel <NEx=440.,NEy=70.> {
  parent => <-.UImod_panel;
  label = “Threshold Module controls:”;
  y = 10;
  width = 200;
 };
};
```

UI Macro (3)

- ❑ And appears as:



User Macro

- Encapsulate all components into a user macro
  - common use will be to process Field and have a user interface
  - main use is application building, not component development
- Macro composed of:

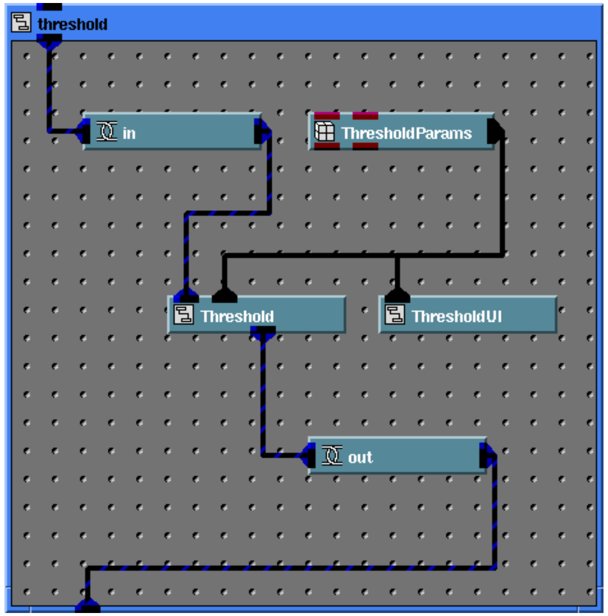
```
macro threshold {
 link in <NEportLevels={2,1},NEx=100.,NEy=50.>;

 IAC_PROJ.Threshold.ThresholdMods.ThresholdParams
  ThresholdParams <NEx=300.,NEy=50.>;

 IAC_PROJ.Threshold.ThresholdMacs.Threshold Threshold <NEx=100.,NEy=250.> {
  &ThresholdParams => <-.ThresholdParams;
  in => <-.in;
 };
```

User Macro (2)

```
 IAC_PROJ.Threshold.ThresholdMacs.ThresholdUI
 ThresholdUI <NEx=300.,NEy=250.> {
  &ThresholdParams => <-.ThresholdParams;
  UImod_panel {
   title => name_of(<-.<-.<-);
  };
 };

 link out <NEportLevels={1,2},NEx=100.,NEy=400.> => Threshold.out;
};
```

User Macro (3)

- ❑ And appears as:



Example Application

- ❑ Always useful to show new modules and macros in use
  - use common data (if possible use installation data)
  - keep it simple
- ❑ Generate example by building an application
  - then save application as normal
  - add V code to project files

Task 12

Refer to Exercise 7:1 in the Workbook

- ❑ Download and install the IAC project files
- ❑ Read the Project Organisation Guidelines
- ❑ Create a module OverlayImages that will combine two images using a transparent colour in the second image