## Jefferson TDR

Section 3 : Objects
Jamie Doornbos
9/17/97
Revision 1.

# Chapter 3 : Objects

**3.1 Types and Instances**
**3.2 Type Resources**
**3.3 Instance Data**
**3.4 Object Placement**
**3.5 Containment**
**3.6 Multi-tile objects**
**3.7 Headlines**
**3.8 Save and Load**
**3.9 Cursor Objects**

## Introduction

**Resource Files.**
Resource files are containers for binary data chunks known as resources. Resources can be any length, and any number of them can be in a file (up to integer and file system limits). Resources are specified by a four character type, 16 bit integer id, and an optional name of up to 255 characters. Names and ids must be unique.

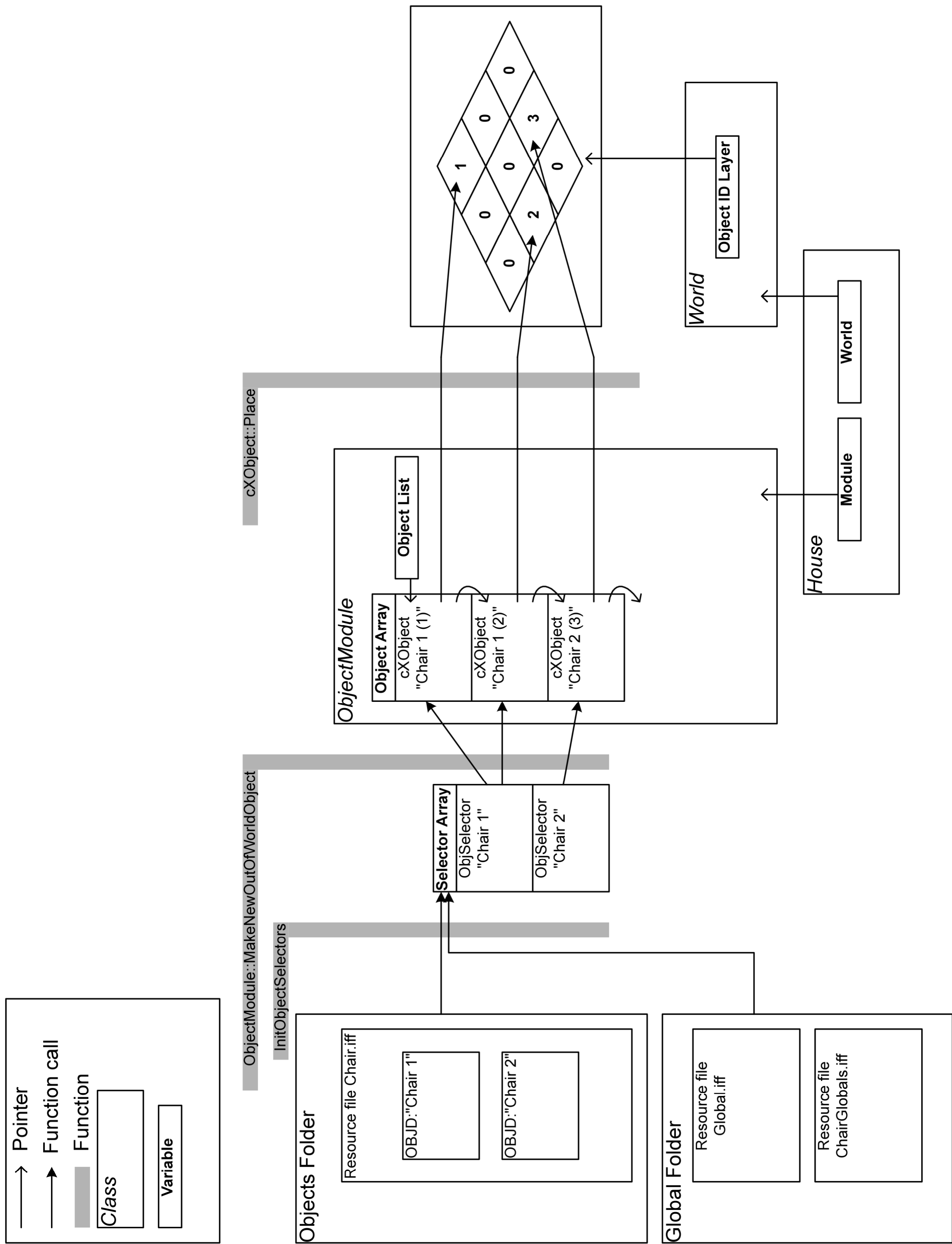
**Objects Folder.**
This folder resides in the TDSScen folder and contains resource files. At initialization time, all files in the objects folder are scanned for object definition resources, type 'OBJD'. A file may contain any number of these definitions. The definition resources are equivalent to the "object type". For each one found, a runtime type specification structure, ObjSelector, is built. From the ObjSelector, the ObjectModule can create a run-time instance of the object, which can then be placed in the house or yard. The final product is an instance of the C++ class cXObject.

**Plug-in Objects.**
A particular type of object in Jefferson, such as "TV", or "Stove", is fully specified by a collection of resources from its file, which are in turn specified by its definition resource. From these resources, many things are determined about an object, among them how it is to be drawn and how it is interacted with.

**Global folder.**
Objects are allowed to use behaviors ('BHAV' resources) from 3 different files: the private, the semi-global, and the global. The private file is the file from which the object's definition ('OBJD' resource) came. The other two reside in the Global folder. The global file is a special resource file which has behaviors that may be used by all objects. Each object also may optionally use behaviors from one semi-global file, which is specified at the object file level.

**Figure**
Diagram of the process of going from an Objects Folder containing two chair types to having chairs in the world. The details


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**





**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

## 3.1 Types and Instances

For each object type, there is one and only one corresponding OBJD resource in a file in the Objects folder. An OBJD resource is a type definition structure, containing enough information to fully instantiate a new cXObject, the runtime object base class. Various routines in ObjectSelector.cpp scan the objects folder and find all the type definitions, and for each one, builds an 'ObjSelector' struct, which contains certain static resources for the type. This process takes place once at the start of the game. TBD: Dynamic addition of new object definitions and selectors without restarting. The class 'ObjectModule' is a container for object instances and also serves as a factory to produce instances of cXObject.

### 3.1.1 Selectors

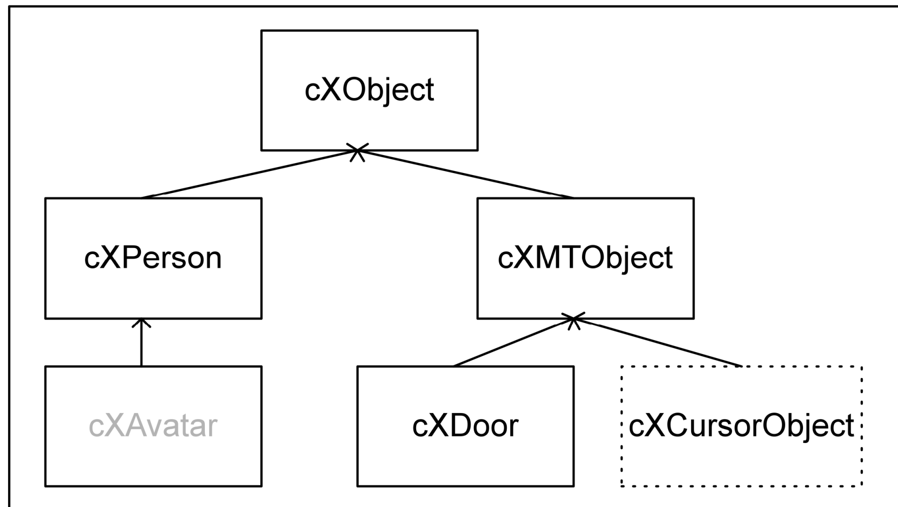
An ObjSelector is the runtime struct for stashing various type-related data which do not change over the span of the game, including a pointer to the OBJD resource and iResFile objects. To build the table of all selectors, each file in the Objects folder is analyzed for OBJD resources using the iResFile interface. For each OBJD resource, a selector is built and added to the table. Thus selectors are one-to-one with OBJD resources (except for disabled types, see 3.2.1.15).

To build a selector, the following steps are performed:

- 3 iResFile objects are created and opened:
  - private, where the OBJD came from;
  - semi-global, see 3.2.2, can be NULL if the object does not need it; and
  - global, the same for all objects : TDSScen/Global/Global
- A cRenderer object is made from the private file. Class cRenderer is a sprite management class and a wrapper for blitting that determines clipping and which blitter to use. (see also graphics, ???).
- A LayerBehavior object is made from all 3 files. A LayerBehavior is a subclass of Behavior, which is used as the instruction fetcher for the Tree Simulation. (see Virtual Machine, ???)
- An Animtable is loaded if the animTableID field of the OBJD is non-zero (see 3.2.1.13 and 3.2.9).
- The name of the OBJD resource is cached as the object's name. TBD: How to name people and objects dynamically.
- The OBJD resource is locked and a pointer to it's memory is assigned to a field in the selector.

### 3.1.2 Modules

An ObjectModule's primary function is as a container and factory for cXObjects. It performs the function of instantiating a cXObject from an 'ObjSelector', including picking a unique runtime object ID, which is used in the world's object layer and in various state variables. The C++ class hierarchy for objects is demonstrated in the figure below. The class cXAvatar is no longer used but is still in the hierarchy, so is included in the picture. The cXCursorObject class is shown with a dashed border because it is not created or returned by the ObjectModule, but instead the class has its own static methods for generating a cursor object (see Cursor Objects, section 3.9).





Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

The ObjectModule also knows how to kill an object, using recursion on multi-tile objects if necessary. The Module also contains a list of objects that were requested to be killed during a simulation tick, which it later kills in the PostSim routine. It also drives the saving and loading process for objects (see Save and Load, 3.8), and other functions that do things to all the objects at once.

### 3.2 Type Resourcess

There are a multitude of things to be specified for instantiating an object, as well as for displaying and interacting with the object. In Jefferson, these are consolidated in OBJD resources on disk, and in selectors at runtime. By the time everything is loaded properly, there will be an object with graphics, behavior, sounds and animations.

#### 3.2.1 OBJD Resources

An OBJD resource contains all the information necessary to construct and initialize a runtime instance of the object (except for the semi-global file, see 3.1.2). It is just a sequence of integer values with various interpretations. These values are edited in the tool "Edith" in the Object Definition window. The end of the data is padded with ample zeroes to allow for future additions without hassle of version updating (as long as zero is suitable default value for new fields). Currently there are 56 16-bit zeroes at the end of the structure.

**3.2.1.1** version – the current version of OBJD. Has not changed for a while because the zero padding trick.

**3.2.1.2** stackSize – how many levels are allocated in the tree simulation's stack for this object. TBD: this may end up being more dynamic.

**3.2.1.3** baseGraphic – the id of the DGRP resource which corresponds to graphic number 0 (see Draw Groups, 3.2.8).

**3.2.1.4** numGraphics – the supposed number of DGRP resource allowed to be used by this object (see also kGraphicNumber, 3.3.2.5.1). Currently not well-enforced.

**3.2.1.5** initBhav – the id of the initial tree the object is to execute. This tree is called outside of the regular simulation to initialize the object. This is necessary because some important fields are needed while the object is being placed, and the sim for the object is disabled during placement. Currently the initialization is assumed to take place in the first tick of the init tree, and the rest is the real 'main' tree. TBD: init tree and main tree may need to be separated so the init can be run separately without disturbing current execution state. This is easily achievable with a new "mainTreeID" field in OBJD.

**3.2.1.6** toolbarPict – the id of the PICT resource used to draw an icon of the object in the pull down object menu. This is not used, as we currently have text in that menu.

**3.2.1.7** treeTableID – the id of the TTAB resource which is the interaction table for this object (see also Tree Tables, 3.2.6).

**3.2.1.8** personalityID – the id of the PERS resource which is the personality for this object. This field is ignored for non-people types. Personality has been on the back burner and is not really used, but the Personality class may at some point be reused and salvaged.

**3.2.1.9** type – one of several values: kUnknown, kFood, kPerson, kContainer, kFurniture, kStructure, kAnimal, kSimType, kDoor, kMouseType, kUserAvatar, kInternal. The type field is used to help determine the C++ type of object to construct. Currently, all of these types map to cXObject except for: kUserAvatar maps to cXUserAvatar (not used); kDoor maps to cXDoor; kMouseType (for cursors such as the wall needle and the object under-square) maps to cXCursorObject; kPerson maps to cXPerson. Another exception is that any type can map to a cXMTObject if the multi-tile fields are specified (see 3.2.1.10, 3.2.1.11 below and Multi-tile objects, 3.6).

**3.2.1.10** masterID – the id that binds multi-tile objects together. If zero, object is single-tile. The binding should be scoped to the file of the definitions. TODO: this is not actually scoped to the file today. Thus a multi-tile object can but should not be made up of objects from different files.

**3.2.1.11** subIndex – this is an encoded x,y offset for multi-tile objects. See Multi-tile coordinates ??? in Chapter 2 for an explanation of the coordinate system. This field


Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

is ignored for single-tile objects. The top 8 bits (most significant) are the x offset and the bottom 8 are the y offset. Both x and y must be greater than or equal to zero. If subIndex==-1, then this is the master type for the multi-tile object. When the object module is requested to instantiate a master type, it actually instantiates all the other types with the same master id, and links them up (complicated piece of code). Requesting an instance of a part of a multi-tile object (masterID!=0 and subIndex!=-1) is an error. TBD: the subIndex field is hard to type; separate x and y fields needed. The mapping to and from master and sub objects is isolated in global functions GetSubTileSelector and GetMasterSelector in ObjSelector.cpp.

**3.2.1.12** dialogID – the id of the DITL resource to display when the user requests it. This is not used anymore. TBD: may be reinstated if we decide to use dialogs.

**3.2.1.13** animTableID – the id of the STR# resource (or CST) that this object uses as its animation table. No animation table is loaded if this is zero. (see also Animation Table, ???).

**3.2.1.14** guid – Globally unique identifier. Needed for saving and loading objects. Generated by code in StubObject.cpp, called by TDSBuildObject when used with -tmpl option.

**3.2.1.15** disabled – can be set in the editor to keep an OBJD from being used in the game.

**3.2.1.16** portalTreeID – for door objects, the id of the behavior tree to run when someone needs to walk through the door. Ignored for non-doors.

**3.2.1.17** price – cost of the object. This cannot be changed over the lifetime of the object.

**3.2.1.18** bodyStringsID – for people, the id of the STR# (or cst) resource to use as the body strings. Strings in this string group correspond to an enum which specifies which suits and skeletons a person uses, and also reg. points, etc. Ignored for non-people types. (See also People, chapter 4.)

**3.2.1.19** slotsID – id of SLOT resource to use for this object's slots. A slot resource is a list of binary slots.

**3.2.1.20** headLinesID – the base id of the SPR# resources to use for this objects imported headlines. A behavior tree can set the headline to a local headline index. Index 0 corresponds to this sprite (see also Headlines, 3.6).

**3.2.1.21** eventTreeID – id of behavior tree to be run when an old-style animation event occurs. No longer used or working.

**3.2.1.22** selfModTreeTableID – id of TTAB resource used as the reflexive interaction table. The trees in this table appear on a menu when the object is clicked on with the reflexive tree tool.

### 3.2.2 GLOB resources

The GLOB resource is just a string (length prefixed) that specifies the name of the semi-global file to use when instantiating a Behavior for this object. This is expected to be relative to the Global directory. A semi-global file contains behavior trees with ids in the semi-global range (8192 and up). The GLOB resource is one-per-file. An indeterminate one will be used if there are several. TBD: Does a GLOB resource need to be specified in each object definition?

### 3.2.3 BHAV : Behavior Trees

The BHAV resource is the lean and mean array of instructions. Its resource id is also the "tree id" and is used to reference the resource from a variety of places. These resources are stored in possibly 3 files for each object. The file for a given tree id is determined by the range it is contained in. This id-to-file mapping is done in class Behavior and class LayerBehavior. Class Behavior also takes care of locking down all the resources and parsing the array to return a simple instruction line. (see also chapter 7 ???).

### 3.2.4 POSI : Positional resources

The POSI resource is the saved state of the editor window for a BHAV. It contains the position and size of each node in the editing window, as well as comments, and extra nodes that do not appear in the behavior itself. For a given BHAV resource, the corresponding POSI resource is just the one in the same file with the same id. POSI resources are only loaded when a tree is to be viewed, such as in trace mode, or in Edith.

### 3.2.6 TTAB : Tree Tables


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

A tree table stores a variable number of interactions. An interaction consists of a check tree id and an action tree id and satisfaction levels for each motive. The satisfaction levels are compressed to only give the non-zero values in the resource. At runtime, these resource are loaded by the ResTreeTab class. A subclass, ObjTreeTab, is contained in each object instance. Each object instance gets its own copy of the static tree table for the type, though currently the values are not saved. TBD: Currently, each object has a copy of the type's tree table, but do they really need it? (see also: Object Interactions ???)

#### 3.2.7 SPR# : Sprite List

A single sprite is identified by its file, its resource id, and its index. An SPR# resource contains version info and an offset table. An individual sprite is found using this table to add to the address of the beginning of a resource. Each sprite has header info and compression data. The header has a bit mask denoting which types of tokens it contains, and a bounding box. The compression data is tokenized to do run length, clear space, patterns, and shades. (see also TDSBuildObject in 17.4.1 and Blitters in 10.2).

#### 3.2.8 DGRP : Draw Groups

A draw group contains enough information to render a single frame of an object in any of 4 rotations and 3 zooms. This is done with a list of draw lists. Each draw list has a bit mask denoting which rotation the frame is valid for, and an integer value denoting the zoom. To select a draw list, a rotation and zoom are specified. When drawing an object, the object's rotation is composed with the view rotation, and then, with the current viewing zoom, the appropriate draw list is selected. Within each draw list is a list of tokenized "draw items". The different types of tokens that can occur are:

1. Sprite: id and index with x, y pixel offset and flipping flags.
2. Top Object: x, y offset of where to draw the target object. Not used anymore because it does not encode a depth offset for the sprite. This has been replaced by slots, which do store depth information.
3. Body Token: draw this object's old-style body (not used anymore)
4. Slot Object Token : (x, y, alt) 3D fixed-point coordinate (see Slot Coordinates in Chapter 2 ???), in object's frame of reference. A slot index denoting which slot object to draw there.

Slot Object Tokens can currently be left out of the draw group entirely for slots that do not change positions from frame to frame. If no slot tokens are in the draw group , all the object slots will be drawn using the coordinates in the slot itself after the draw group has been drawn. (See also Slots, 3.5.1)

#### 3.2.9 STR# (or cst) : Animation Table.

STR# is a list of strings. The animation table uses an array of strings to refer to vitaboy skill names. The animation table is loaded from the three resource files of an object by taking skill names from the private file first, then filling in any blanks with the semi-global and then the global. [ TBD: do we really need the cascaded animation tables? It seems like objects will either know what their animation is or not. ] The person's animation table has standard animations that all people have, such as stand and sit. In this way, an object can tell a person to do their own special categorical animation without knowing what it is. (See also Primitive Animate New:???) An object's animation table contains animations for interacting with that object. Currently, no objects have their own animations. All are stored in the people. [TBD: How to get animations for multiple skeletons transparently into the object's animation list. This can be done by having a new skeleton parameter in the animation table's GetEntry routine. Then to load an animation table, a resource id for each skeleton type must be provided, which would just be new fields in the OBJD. ]

#### 3.2.10 Sounds : FWAV resources

Sounds are specified by the id of an FWAV resource. An FWAV resource is currently just a c-string that specifies the name of a sound file. Currently all sounds are global and the name is specified relative to the sounds directory in TDSScen. TBD: how to represent local sounds. TBD: how to play sounds from an object resource file.

### 3.3 Instance Data

Basically these are the members of the cXObject class


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

**3.3.1 Properties. Members that are the same for the lifetime of the object.**

**3.3.1.1** ObjectModule* fModule – the ObjectModule that this object belongs to.

**3.3.1.2** cXObject* fNext – next object in the module's object list

**3.3.1.3** Sint16 fID – the id of the object. Used in the world grid (see World, chapter 2), and in the module's object table. Ranges between 1 and ObjectModule::kMaxObjects. ID 0 is null.

**3.3.1.4** Definition* fDef – the object definition that was used to create this object. This is a pointer to an OBJD resource which has been locked down.

**3.3.1.5** ObjTreeTab* fTreeTable – the tree table of this object. Used when a person autonomously looks for interactions or when the user clicks to see a menu. The trees in this table are assumed to be run by the interacting person with the targeted object in the stack object field (see also Virtual Machine: ???). The advertisements in the table can be modified by the behavior trees, but are not currently being written to disk. TBD: No current objects are statically modifying tree table advertisements, should it be allowed?

**3.3.1.6** ResTreeTab* fSelfModTreeTab – the reflexive tree table of this object. Use when the user clicks on an object with the reflexive tree tool. The trees in this table are assumed to be run by the object, and are currently just interrupted onto the object's stack and allowed to execute on the next sim tick. Trees referred to by this table are primarily for debugging or setting properties.

**3.3.1.7** ObjSelector* fObjSel – the selector used to create this object.

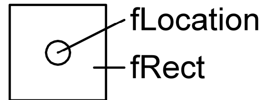
**3.3.1.8** cVoice* fVoice, cChannel* fChannel – Formerly used in Jacques' feature: X People talking on the Mac. Now is not used and stubbed out.

**3.3.1.9** RTSlotList fSlots – the list of slots for this object (see also Slots, 3.5.1).

**3.3.2 State. Variables that change through simulation or other means.**

**3.3.2.1** FTileRect fRect – the rectangle of the object in fixed world fractional tile coordinates.

**3.3.2.2** FTilePt fLocation – the location of the object. Always at the center of fRect




**3.3.2.3** RelMatrix fInstMatrix – the relationship matrix keyed off of the ids of people in the module (TBD: a new matrix or possibly the same matrix that uses a yet to be defined "affinity" value as the key.)

**3.3.2.4** SInt16 fSimEnabled – whether or not the object is simulated. Currently always on except during placement.

**3.3.2.5** SInt16 fData[kNumData] – Data Array. Currently kNumData is 36. Accessed directly by behavior trees and compiled code. An asterisk(*) means the value is only used in people.

3.3.2.5.1 kGraphicNumber – the current graphical frame that is rendered for the object. Ranges from 0 to numGraphics-1 (see numGraphics, 3.2.1.4).

3.3.2.5.2 kDirection – the current direction the object is facing (0-7). 0 is north, back and to the right in an unrotated world (see chapter 2, Object Direction).

3.3.2.5.3 kColor1 and kColor2 – the palette entries to use when blitting a shaded sprite as part of the object. TBD: do we need shaded sprites?

3.3.2.5.4 kPattern – the pattern number to use when blitting a patterned sprite as part of the object. TBD: do we need patterned sprites?

3.3.2.5.5 kHeight – the nominal height of the object. Was used to determine where to draw the hilite arrow above objects, but currently not used.

3.3.2.5.6 kRouteID* – the id of the route that a person is following.

3.3.2.5.7 kIndirectID – previously used in some primitives for accessing fields in another object. Currently not used.

3.3.2.5.8 kFlagField1 – set of various flags for the object.


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

- 3.3.2.5.8.1 kCanWalkOver – whether or not an object can be stepped on as though it were the ground.
- 3.3.2.5.8.2 kCanWalk – whether or not the object is mobile. Should be set for all people all the time, since it is asserted on in primitive handler Goto Relative.
- 3.3.2.5.8.3 kPreviouslyFound – whether or not the object has been found in a behavior tree search. Cleared by primitive "Start Search" and set by "Find Action [new]". Field is not guaranteed after a tick has passed, since other trees could be searching.
- 3.3.2.5.8.4 kOccupied – whether or not to allow interaction with the object. Set by behavior trees. Tested by code and trees.
- 3.3.2.5.8.5 kNotified* – Whether or not the user clicked when the person was in a non-interruptable "Idle For Input" primitive (See also Idle For Input Primitive, ???).
- 3.3.2.5.8.6 kRoutingInterruptable* – Whether or not the goto relative primitive will fail immediately if an action is in the action queue. TBD: Can be used to force a routing action to terminate. (See also Goto Relative: ???)

3.3.2.5.9 kAnimID* – Currently is always 0 or 1. 0 means no object animation is playing. 1 means that it is (see also Primitive Animate New: ???). Formerly was used (and is still in #ifdefs) for the resource id of the animation being played on the old-style person bodies.

3.3.2.5.10 kAnimFrame – Currently not used. Formerly used in the old style body animation to be the current frame.

3.3.2.5.11 kObjectID – the id of the object. This is a duplicate of member fID so behavior trees can access it.

3.3.2.5.12 kOldTargetID – not used directly. Only here as a placeholder for behavior tree references. Mapped by primitives to the object id in the default slot (slot 0).

3.3.2.5.13 kWallPlacementFlags – bit mask for where the object can be legally placed with respect to walls.

- 3.3.2.5.13.1 kWFAnywhere – walls do not affect placement.
- 3.3.2.5.13.2 kWFInFront – wall required in front of object
- 3.3.2.5.13.3 kWFRightSide – wall required on right side of object
- 3.3.2.5.13.4 kWFBehind – wall required behind object
- 3.3.2.5.13.5 kWFLeftSide – wall required on left side of object
- 3.3.2.5.13.6 kWFNoCorner – cannot be placed in a corner
- 3.3.2.5.13.7 kWFCornerOnly – can only be placed in a corner

3.3.2.5.14 kSlotID – the index of the slot that this object is contained in. Only supposed to be set by the object moving code.

3.3.2.5.15 kFamilyNumber – the family number of this object. 0 means no family, like visitors.

3.3.2.5.16 kCounter1 – generic counter. Not currently used.

3.3.2.5.17 kTrapCount* – state variable owned by primitive goto relative. Shares the same address as kCounter1. If 0, the person is not currently involved in routing. If greater than 0, it is the number of tries the person has left to get the route right. If less than 0, it is the object id of the portal object that failed. This is so cascaded routes can tell when they have failed. TBD: may need mre state variables for route planner, possibly some stack locals, using the idea of "putting the primitive on the stack".

3.3.2.5.18 kRoomCompDelay – doors only. Shares same address as kCounter1. Not currently used. Formerly it was decremented each tick by cXDoor::Simulate, and when it reached 0, the tree table of the door was recomputed to reflect all the objects in the opposite room.

3.3.2.5.19 kAge – age of the object. Currently not used explicitly.

3.3.2.5.20 kGender – gender of the object. Only used by behavior trees.


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

3.3.2.5.21 kTreeTableEntry – current tree table interaction being examined. Set by certain primitives like “Find Action New” (and “Expression”, which can set any data field), and used by Primitive “Gosub Found Action” to invoke the interaction it refers to.

3.3.2.5.22 kSearchRadius – currently not used. Formerly used to denote how far the object could “see” when searching for other objects.

3.3.2.5.23 kSpeed* – how fast a person walks in 8.8 fixed point fractional tile increments per tick. I.e. a value of 0x0100 corresponds to 1 fractional tile per tick, or 1/16 of a tile, or 3/16 of a foot.

3.3.2.5.24 kRotationSpeed – how much to increment or decrement the object’s direction when rotating. Typically this is 2 for objects and 1 for people.

3.3.2.5.25 kCounter2 – not used

3.3.2.5.26 kRouteCount – shares address space with kCounter2. Not currently used. Formerly a step counter for routing.

3.3.2.5.27 kUseCount – timer for when an object will be made available for interactions again. Currently is being set by behavior trees and decremented in the Simulate routine, but is not being checked by the “Find Action New” primitive to disallow interactions.

3.3.2.5.28 kContainerID – object id of the object that contains this. Supposed to only be set by the object moving code.

3.3.2.5.29 kWeight – representation of the object’s weight. Use to determine if an object is too heavy to place on another object.

3.3.2.5.30 kSupportWeight – how much weight the object can contain. Used to determine if another object can be placed on this one.

3.3.2.5.31 kRoom – the room of the object. Always equal to the value in the room map at the object’s location. Set when object is placed or when rooms change.

3.3.2.5.32 kRoomPlacement – value denoting where an object can be placed with respect to rooms. Takes on one of kRmPlAnyRoom, kRmPlOutsideOnly, kRmPlInsideOnly.

3.3.2.5.33 kHidden – whether or not the object is hidden. Supposed to only be set by “Show Hide” primitive.

**3.3.2.6** SInt16 fAttrs[kNumAttr] – Attribute Array. Currently kNumAttr is 8. Accessed directly by behavior trees through expression primitive, but not at all by compiled code. Used within object types to store state information that does not need to interact with compiled code.

**3.3.2.7** SInt16 fTemp[kNumTemp] – Temp Array. Currently kNumTemp is 8. Accessed directly by behavior trees using expression primitive. Used to pass values around, but not guaranteed to keep its value across a sim tick.

**3.3.2.8** SInt16 fIterations – this is duplicated from TreeSim. This is an error.

**3.3.2.9** SInt16 fDirInc – not used anymore. Used to be a seed value for route planning.

**3.3.2.10** RTSlotList fSlots – The list of slots for this object. Some slot values change as part of the state, such as the object id of a contained object slot. That is why slots are mentioned in properties and state.

**3.3.2.11** Boolean fIsDirty – indicates if the object need to be drawn. Gets set when the object’s Dirty method is called, but is never cleared and never tested. TBD: this field may later be used by the HouseViewer to determine if the object needs to be drawn.

**3.3.2.12** Int fHiliteMask – a mask indicating the current hiliting of the object. Passed down to the blitters and vitaboy to render objects in different hilite states. Currently the hilite flags are kHilited, kSelected, kCanInterrupt. The selected person and only the selected person has the kSelected flag set. The tools and the mouse object set the kHilited flag to show an object is being placed. The kCanInterrupt mode was formerly set when people were busy, but is currently not used. TBD: other hilite modes?


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

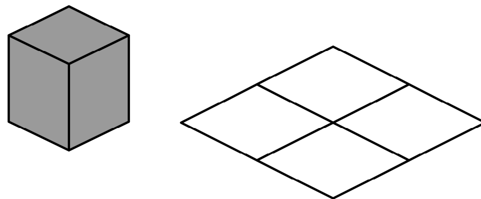
**3.4 Object Placement** Objects can be located in a number of ways. Starting with objects in the grid, there is an n-ary tree of containment, where the n is limited by the number of slots in an object. The root node is an abstract node with all grid objects as its leaves. Other nodes are objects and can have an object id in each of its statically defined slots. The position of an object in this tree is determined by its data fields, kContainerID and kSlotID, and its location. The interface to object placement is this:

```
virtual void Pickup();
virtual Boolean CanPlace(FTilePt &newLoc, cXObject* ontop, Int slotNum);
virtual void Place(FTilePt &loc, cXObject* ontop, Int slotNum);
```

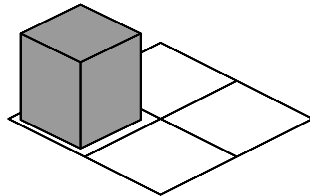
If the ontop parameter is non-null, the location is ignored and the 'this' object goes into a slot in 'ontop' object. Otherwise, the object is placed directly on the grid at the specified location.

**3.4.1 Positions.** An object has three possible states for its position: Out Of World, Directly on Grid, and Hierarchically Contained. TBD: This scheme may change to accommodate multiple sibling objects on a tile.

**3.4.1.1 Out Of World** This is the state achieved by the polymorhic routine "Pickup". An out of world object has an invalid fLocation and is contained in no other objects (fData[kContainerID] = 0). Pickup has no effect if the object is already out of world.

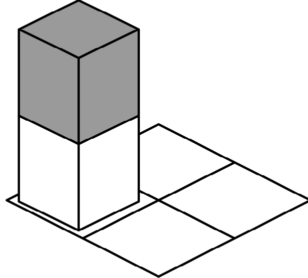


**3.4.1.2 Directly on Grid** The object is in the grid: its id is in the world object layer at fLocation (see World: Chapter 2). The container id (fData[kContainerID]) of the object is 0. The object may, in this state, have other objects contained in it. Objects that can be walked over and multi-tile objects must lie directly in the grid (or out of world).




**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

**3.4.1.3 Hierarchically contained** The object's id is in a containment slot of some other object, whose id is in the container id field. The location is exactly equal to the containing object's, except when the container has the "can walk over" attribute, in which case the location can be anywhere on the tile. Note: When more than one object occupies the same tile, one must be contained in the other. [ TBD: we may need a different scheme to simplify this relationship. One idea is to allow a sibling relationship of objects, in which case multiple objects could reside in any possible position. Another idea is to create a new type of position state in which the location is valid, but the object's id is not entered in the object layer. ]



**3.4.2 Criteria for placement directly on the grid** For a given location, placement of an object there is subject to the following conditions:

1. If the location is out of the world's bounds, the object cannot be placed.
2. If the object has non-zero wall placement flags and the desired location is not in the exact center of a tile, the object cannot be placed.
3. If the wall configuration at the desired tile does not match up with the object's wall placement flags, the object cannot be placed (see 3.3.2.5.13, kWallPlacementFlags).
4. If the room of the tile does not match the room placement flags, the object cannot be placed (see kRoomPlacement, 3.3.2.5.32).
5. Only one object that can be walked over can reside on a given tile, and it must always be in the grid.
6. If the object cannot be walked over, and another object is in the grid that cannot be walked over and is not a part of the given object, the object cannot be placed.
7. If the object cannot be walked over, and another object is in the grid that can be walked over, the object cannot be placed if it cannot be placed on top of the other object (see next section).
8. If the rectangle of any object in one of the 8 adjacent tiles overlaps the would-be new rectangle of this object, and the other object is not part of the given object, the object cannot be placed. TBD This may need to be changed to accommodate for interactions that take place closer than one tile away from an object.
9. If the object's would-be rectangle intersects with a wall, it cannot be placed.

**3.4.3 Criteria for placement on top of another object.** An object can be placed in another object's slot subject to the following conditions:

1. The slot with that number must exist.
2. If the bottom object cannot be walked over and the weight of the object exceeds the support weight of the bottom object, the object cannot be placed.
3. If the object can be walked over, and the bottom object is not part of the object, it cannot be placed.
4. If an object is in the slot and is not a part of the given object, it cannot be placed.

## 3.5 Containment

**3.5.1 Slots.** Each slot has a statically defined 3D coordinate in slot coordinates (see coordinates, chapter 2). This is used only when rendering. The coordinate may be overridden by draw group slot items when necessary (see 3.2.8, Draw Groups). Each slot has a type field that is one of the following: kContainedObject, kHeadline, kHandle.


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

**3.5.1.1 Headline slots.** The following fields are used (see also Headlines, 3.6):

1. SInt16 headlineSpriteID – the sprite id of the headline to draw here.
2. Boolean headlineGlobal – whether or not the sprite id is a global sprite or a local.
3. SInt16 headlineFrame – which frame of the headline is being drawn (fixed point with resolution "frame skip rate", see headlines section).
4. SInt16 ticksLeft – how many more sim ticks to draw it for, decremented each tick.
5. TBD: we may need, instead of "headlineGlobal", another field specifying which object or cRenderer this sprite belongs to. This would be needed to have an object import special headlines to put over people, which would be really cool.

**3.5.1.2 Contained Object Slots.** An objectID field is used to hold the id of the contained object. This object's location, container id and slot num must match up as described below. TBD: A containment slot may have more data that determines how the slot is to be entered and exited and how the person idles in the slot.

**3.5.1.3 Handle Slots.** This is the slot which governs the relative origin of a contained object to that of its slot's origin. All objects have an implied handle at 0,0,0, which is used if no handles are available or if the handle number is zero. TBD: a handle number field should be added to the data fields. An object can have several handle slots. These determine the possible ways an object can be held. This is currently not used in any objects, but some code exists to offset the rendering origin of contained objects.

**3.5.1.4 New slot type: routing slots.** TBD: These would be generic slots used to find a location for a person to route to and specify a registration for running animations.

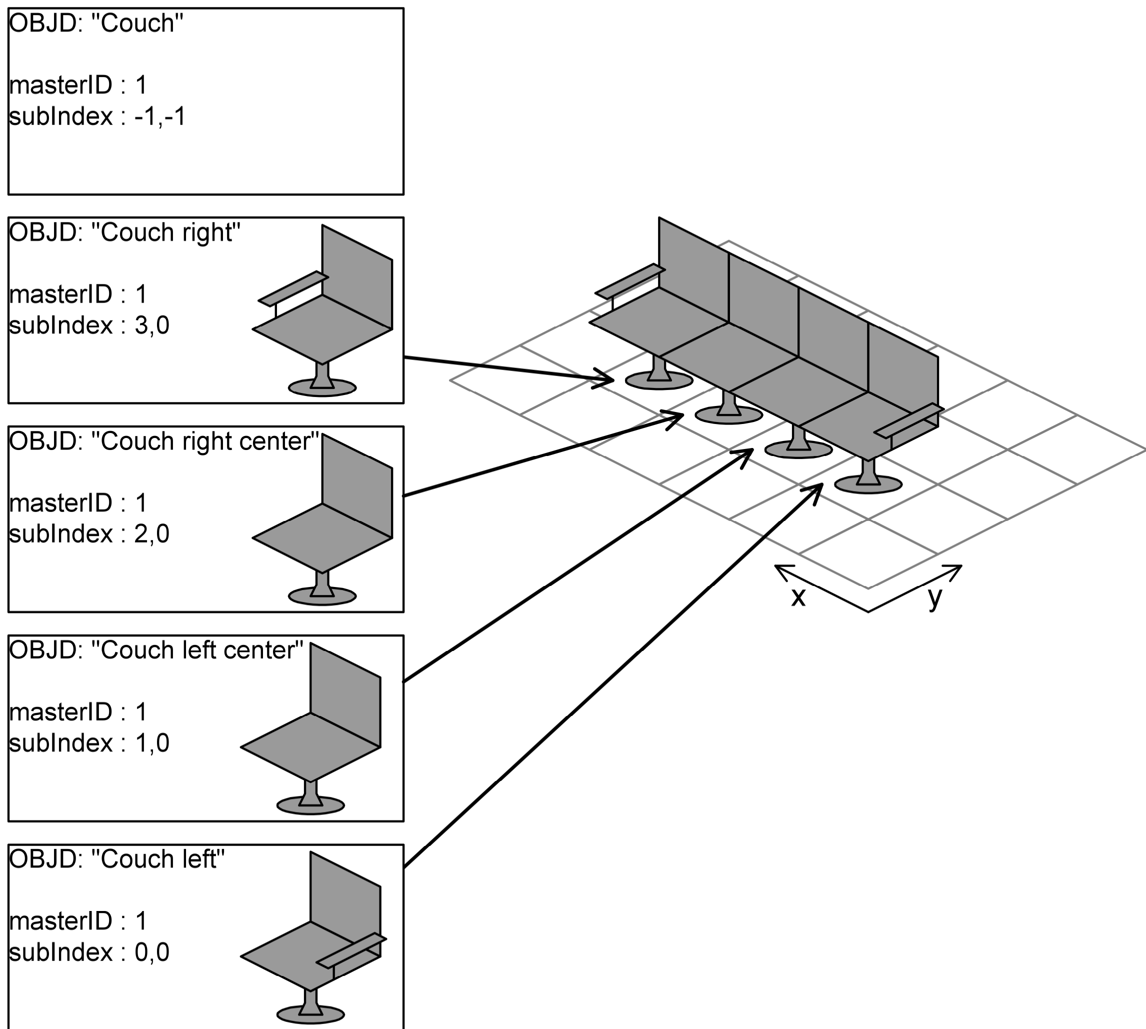
**3.5.2 Interaction.** TBD. Interaction with contained objects is a challenging problem which has not really been figured out. Complications arise in 3 main areas, relating to the complexity of plug-in objects:

1. *Increased complexity of behavior trees.* Behavior trees must adjust the interaction behavior to account for the containment environment of the object. They must test different variables to determine "inside" or "on top of", as well as "height".
2. *Increased complexity of content.* An object comes with its own animations to visually represent the interaction. If the object is contained in or on top of another object, the relative height of the person and the object can be at several different values. (TBD: It is fairly clear that we will define certain standard heights to help tract the problem.) Each possible relative height multiplies the number of animations necessary to represent the full range of interaction. One solution is on the design side. We could limit the interactions on an object by object basis to a very small set of heights, maybe only one height. Along with an "out of hand" interaction as a backup, in which the person would just pick up the object before interacting with it, game play would not be unduly limited.
3. *Increased complexity of user interface.* One problem is representing all the different containment positions to the user wishing to put something in the container. E.g. putting something in a specific location on the stove, and listing all possible places. One idea is to map mouse clicks to a certain slot or coordinate on the object, and send the result to the interaction tree. A simpler problem is when to concatenate the interactions of a contained object. An easy solution comes to mind since check trees will at some point have the capability of hiding their interactions, or making them invisible. Contained objects could invalidate their interactions appropriately. TBD: other problems.


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

### 3.6 Multi-tile objects

This figure demonstrates how a sample multi-tile Couch object would be constructed out of component objects.




**3.6.1 Definition and positioning.** Multi-tile objects are conglomerates of single tile objects bounds together by a "master id" field in the Definition structure (see "masterID", 3.2.1.10). The relative positioning of the objects is done through another field, the subIndex, which has unsigned x, y offsets (see "subIndex", 3.2.1.11). A special -1 value in this positions denotes the "master" selector, which is never actually instantiated. It is a placeholder selector by which multi-tile object instances are requested from the ObjectModule. Other than when the user is moving a multi-tile object around, each component object is exactly the same as a single tile object with respect to draw groups, tree tables, behaviors, etc. Objects can share the same resources, but each Definition must specify the resources it intends to use.

**3.6.2 Creation.** Multi-tile Definitions are special cased in the factory and cause a C++ subclass of cXObject, cXMTObject, to be created. One is created for each selector that exists with the same master id and subIndex in range (0..7, 0..7). TODO: the code to find selectors associated with a master selector currently do not scope the master id to file. This is really a bug.

**3.6.3 Binding.** Instances are bound together in a linked list. Each instance has a pointer to its next multi-tile object, and its leader. The leader has nil in its leader field to denote that it is the leader. Typically the lead object is the one with offset (0,0) because of the way they are created. A primitive exists for getting the next multi-tile object so that the objects' behavior


Confidential and Proprietary Information
May not be reproduced or distributed without the express permission of Maxis, Inc.

can defer interactions or relationships to a designated object (see primitive "Set To Next Object" ???).

**3.6.4 Special placement code.** Multi-tile objects cache the x, y offset values, and adjust them whenever rotation occurs. The tools for object placement treat multi-tile objects as regular objects. Typically, when a single instance is told to do something, such as "Place", or "Turn", it traverses its list of instances, telling each object in turn to perform the base class function, accounting for the x, y offsets. The code for turning a multi-tile object is quite complicated because it must first test all the new locations before actually picking up the object and losing its location state.

**3.6.5 Save and load.** In saving and loading, the special list fields are serialized as their id values. The offsets are not saved, but are regenerated from the saved direction.

**3.6.6 Limitations.** Currently not possible to place multi-tile objects on top of other objects. This is somewhat ill-defined. TBD: do we need to define and implement this? The direction field of mt objects is not really fair game for the behavior trees. Normally objects can rotate themselves from the behavior trees by setting the rotation field, (and updating), but for multi-tile objects, rotation actually changes the position, which is not accessible from the trees. (TBD: do multi-tile object really need to rotate from the trees anyway?) (TBD: do we need to rotate anything from the behavior trees?) (TBD: do we need a rotation primitive?)

**3.7 Headlines.** Headline sprites are attached to slots on an object. The sprites can be local from the cRenderer of an object type, or global from the global cRenderer. The global sprite ids start at constant kGlobalHeadlinesBaseID, and the balloons and icons both come from here. All people have two default headline slots over the head to fit both the surrounding bubble and the icon. The sprites for headlines are normal SPR# resources. Each set of three sprites in the list is a frame in an animated headline, large, meium, and small. Thus the number of frames is the total number of sprites divided by 3. A frame counter is bumped up on each render, and a constant, frameSkipRate, governs how many draws cause a frame change.

**Note on Serialization:**
Utility classes exist for serializing. The ReconBuffer is a serialization buffer that performs different functions depending on its mode: kReading, kWriting, or kCounting. It operates an arrays (possibly of length 1) of bytes, shorts, longs, and floats, and also pascal strings. The ReconObject is an abstract class with a routine to get its type, and a routine to serialize using a ReconBuffer and a version. The ReconBuilder class is the driver. Given a ReconObject and a version, it can pack the object into and out of a block of memory, or save and load the object from an iResFile.

**3.8 Save and Load.** Objects are saved and loaded using the "Reconstitution" suite of utility classes (see Note on serialization ???). Class cXObject has virtual routines, ReconStream, and ReconGetType. A subclass of ReconObject, ReconXObject, is used to load and save the object. In loading, this class first serializes the exact type of the object using ReconSelector, which just used the guid. This can fail if plug-ins have been removed. TBD: how to deal with this. Once the type has been found, it figures out the runtime type, then constructs the instance, initializes it, and serializes it by calling its ReconStream. In the simpler process of saving, ReconXObject first serializes the exact type and then calls ReconStream. The overall process is driven by routines in the ObjectModule: LoadAllObjects and SaveAllObjects. These routines use a ReconBuilder and a ReconXObject to read and write 'Rcon' resources from the save file. The trick is to go in the same order as the object list so that simulation order is preserved and multi-tile objects can save the id of their 'next' multi-tile object. So in saving, the id for a 'Rcon' resource is just the index of the object in the list. Then in loading, the resources are reconstituted in id order from 1 to kMaxObjects. Please see also the Theory of Operations for saving a house file.

**3.9 Cursor Objects.** Cursor objects are special objects whose purpose is to sit underneath regular objects as they are dragged around by the user. The C++ class cXCursorObject descends from cXMTObject, so that cursor objects can easily move as a unit underneath multi-tile objects. A cXCursorObject is created by the static routine, cXCursorObject::MakeCursorObject. This routine is overloaded to take


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**

either an object, or an ObjSelector. In the object version, cursors are created to go underneath each tile of the existing object. For the object selector version, a new object is created from the selector and the object pointer version is called. The cursor object records in a field whether or not the object was created or just picked up. If the object was created, funds are deducted when the object is dropped. Also a parameter to this creation is the selector used to generate the cursor object itself. Currently these selectors are hard-coded by name. The names “Wall Cursor” and “Tile Cursor” are currently used to look up a selector when generating new cursor objects and the program may fail if these are not present. TBD: how and to what extent should the existence of these special object files be enforced? The cursor object also has special routines for positioning an object in a possibly illegal position. That is, the code ignores the normal placement constraints so the object will look as it would look if it could be placed in a given location. The tile cursor object is also expected to have certain draw groups for when the object it contains can be placed and when it cannot be placed. TBD: this may need more visual indication, such as a different hilite mode on the object.


**Confidential and Proprietary Information**
**May not be reproduced or distributed without the express permission of Maxis, Inc.**