

Code

Portions of this code are modified standard behaviors of Macromedia Director

Main Script

```
global gClock
global gState
global gEZLocked
global gOn
global gStopped
global TimeCook
global stopMilliseconds
global clearMilliSeconds

on pressDefrost
  if gEZLocked = TRUE then
    puppetSound "Defrost"
  else --not in EZ mode
    if gOn = TRUE then
      nothing
    else --not cookin'
      if gState = 0 then
        member("MicroTime").text = "00:00"
      end if
      puppetSound "ShortBeep"
      if objectP(TimeCook) then --object exists
        gState = 1
      else --object doesn't exist
        TimeCook = new(script "setTimerMode")
        TimeCook.setTimer()
        gState = 1
        put "new object"
      end if
    end if
  end if
end pressDefrost

on pressMedium
  if gEZLocked = TRUE then
    puppetSound "MediumCook"
  else --not in EZ mode
    if gOn = TRUE then
      nothing
    else --not cookin'
      if gState = 0 then
        member("MicroTime").text = "00:00"
      end if
      puppetSound "ShortBeep"
      if objectP(TimeCook) then --object exists
        gState = 2
      else --object doesn't exist
        TimeCook = new(script "setTimerMode")
        TimeCook.setTimer()
        gState = 2
        put "new object"
      end if
    end if
  end if
end pressMedium

on pressHigh
  if gEZLocked = TRUE then
    puppetSound "HighCook"
  else --not in EZ mode
    if gOn = TRUE then
```

```

    nothing
else --not cookin'
if gState = 0 then
    member("MicroTime").text = "00:00"
end if
puppetSound "ShortBeep"
if objectP(TimeCook) then --object exists
    gState = 3
else --object doesn't exist
    TimeCook = new(script "setTimerMode")
    TimeCook.setTimer()
    gState = 3
    put "new object"
end if
end if
end if
end pressHigh

```

```

on press1
if gEZLocked = TRUE then
    puppetSound "1"
else --not in EZ mode
if gOn = TRUE then
    nothing
else --not cookin'
    puppetSound "ShortBeep"
    if (gState > 0 and gState < 4) then
        TimeCook.addTimer(1)
    end if
end if
end if
end

```

```

on press2
if gEZLocked = TRUE then
    puppetSound "2"
else --not in EZ mode
if gOn = TRUE then
    nothing
else --not cookin'
    puppetSound "ShortBeep"
    if (gState > 0 and gState < 4) then
        TimeCook.addTimer(2)
    end if
end if
end if
end press2

```

```

on press3
if gEZLocked = TRUE then
    puppetSound "3"
else --not in EZ mode
if gOn = TRUE then
    nothing
else --not cookin'
    puppetSound "ShortBeep"
    if (gState > 0 and gState < 4) then
        TimeCook.addTimer(3)
    end if
end if
end if
end press3

```

```

on press4
if gEZLocked = TRUE then
    puppetSound "4"
else --not in EZ mode
if gOn = TRUE then
    nothing
else --not cookin'

```

```

        puppetSound "ShortBeep"
        if (gState > 0 and gState < 4) then
            TimeCook.addTimer(4)
        end if
    end if
end if
end press4

on press5
    if gEZLocked = TRUE then
        puppetSound "5"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if (gState > 0 and gState < 4) then
                TimeCook.addTimer(5)
            end if
        end if
    end if
end if
end press5

on press6
    if gEZLocked = TRUE then
        puppetSound "6"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if (gState > 0 and gState < 4) then
                TimeCook.addTimer(6)
            end if
        end if
    end if
end if
end press6

on press7
    if gEZLocked = TRUE then
        puppetSound "7"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if (gState > 0 and gState < 4) then
                TimeCook.addTimer(7)
            end if
        end if
    end if
end if
end press7

on press8
    if gEZLocked = TRUE then
        puppetSound "8"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if (gState > 0 and gState < 4) then
                TimeCook.addTimer(8)
            end if
        end if
    end if
end if
end press8

on press9
    if gEZLocked = TRUE then
        puppetSound "9"
    end if
end if
end press9

```

```

else --not in EZ mode
  if gOn = TRUE then
    nothing
  else --not cookin'
    puppetSound "ShortBeep"
    if (gState > 0 and gState < 4) then
      TimeCook.addTimer(9)
    end if
  end if
end if
end press9

on press0
  if gEZLocked = TRUE then
    puppetSound "0"
  else --not in EZ mode
    if gOn = TRUE then
      nothing
    else --not cookin'
      puppetSound "ShortBeep"
      if (gState > 0 and gState < 4) then
        TimeCook.addTimer(0)
      end if
    end if
  end if
end press0

on pressMinute
  if gEZLocked = TRUE then
    puppetSound "Minute"
  else --not in EZ mode
    if gOn = TRUE then
      nothing
    else --not cookin'
      puppetSound "ShortBeep"
      if objectP(TimeCook) then --object exists
        TimeCook.addMinute()
      else --object doesn't exist
        TimeCook = new(script "setTimerMode")
        TimeCook.addMinute()
        put "new object"
      end if
    end if
  end if
end pressMinute

on setClock
  if gEZLocked = TRUE then
    puppetSound "SetClock"
  else --not in EZ mode
    puppetSound "ShortBeep"
    gState = 0
    put gState
  end if
end setClock

on pressPopcorn
  if gEZLocked = TRUE then
    puppetSound "Popcorn"
  else --not in EZ mode
    if gOn = TRUE then
      nothing
    else --not cookin'
      puppetSound "ShortBeep"
      if objectP(TimeCook) then --object exists
        TimeCook.presetTimer(315)
        gState = 4
      else --object doesn't exist
        TimeCook = new(script "setTimerMode")
        TimeCook.presetTimer(315)
        gState = 4
      end if
    end if
  end if
end pressPopcorn

```

```

        put "new object"
    end if
end if
end if
end pressPopcorn

on pressBeverage
    if gEZLocked = TRUE then
        puppetSound "HotBeverage"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if objectP(TimeCook) then --object exists
                TimeCook.presetTimer(45)
                gState = 5
            else --object doesn't exist
                TimeCook = new(script "setTimerMode")
                TimeCook.presetTimer(45)
                gState = 5
                put "new object"
            end if
        end if
    end if
end pressBeverage

on pressDinner
    if gEZLocked = TRUE then
        puppetSound "FrozenDinner"
    else --not in EZ mode
        if gOn = TRUE then
            nothing
        else --not cookin'
            puppetSound "ShortBeep"
            if objectP(TimeCook) then --object exists
                TimeCook.presetTimer(530)
                gState = 6
            else --object doesn't exist
                TimeCook = new(script "setTimerMode")
                TimeCook.presetTimer(530)
                gState = 6
                put "new object"
            end if
        end if
    end if
end pressDinner

on pressGo
    if gEZLocked = TRUE then
        puppetSound "Go"
    else --not in EZ mode
        if objectP(TimeCook) then --object exists
            if gOn = TRUE then
                nothing
            else --not cookin'
                puppetSound "ShortBeep"
                put "Go!"
                gOn = TRUE -- lock all responses except stop
                gStopped = FALSE
                sprite(23).pCurrentMinutes = TimeCook.theTimer / 100
                sprite(23).pCurrentSeconds = TimeCook.theTimer mod 100
                sprite(23).pPeriod = 60000 * sprite(23).pCurrentMinutes \
                + 1000 * sprite(23).pCurrentSeconds
                --if stopMilliseconds = 0 then --first run
                sprite(23).pStartMillis = sprite(23).pStartMillis \
                + (the milliseconds - stopMilliseconds) + clearMilliseconds
                --else
                --sprite(23).pStartMillis = sprite(23).pStartMillis \
                + (the milliseconds - stopMilliseconds)
                --end if
            end if
        end if
    end if
end pressGo

```

```

        put "Millisecs right now  :" & the milliseconds
        sprite(23).pActive = TRUE
        put sprite(23).pCurrentMinutes
        put sprite(23).pCurrentSeconds
        put sprite(23).pPeriod
        put "start millis: " & sprite(23).pStartMillis
        updatestage
    end if
end if
end if
end pressGo

on pressStop
if gEZLocked = TRUE then
    puppetSound "Stop"
else
    puppetSound "ShortBeep"
    if gOn = TRUE then --stop timer
        stopMilliseconds = the milliseconds
        put "Stop!"
        put the milliseconds
        gOn = FALSE
        gStopped = TRUE
        sprite(23).pActive = FALSE
    else
        clearMilliseconds = sprite(23).vMillis - sprite(23).pStartMillis
        put "Clear! " & clearMilliseconds
        put the milliseconds
        TimeCook = VOID
        gState = 0
        gStopped = FALSE
        member("MicroTime").text = "00:00"
        repeat with n = 2 to 8 --return buttons to normal
            sprite(n).myState = VOID
            updatestage
        end repeat

        sprite(23).pPeriod = 0
        sprite(23).pActive = FALSE
    end if
end if
end pressStop

on pressEZ
if gOn = TRUE then
    nothing
else
    if gEZLocked = TRUE then
        gEZLocked = FALSE
        put "EZ Unlocked"
        puppetSound "EzOff"
        repeat with n = 2 to 8 --return buttons to normal
            sprite(n).myState = VOID
            updatestage
        end repeat
    else
        gEZLocked = TRUE
        put "EZ Locked"
        puppetSound "EzOn"
    end if
end if
end pressEZ

on doneCookin
stopMilliseconds = the milliseconds
put "done cookin"
puppetSound "DoneCookin"
TimeCook = VOID
gState = 0
gOn = FALSE
gStopped = FALSE

```

```

repeat with n = 2 to 8 --return buttons to normal
  sprite(n).myState = VOID
end repeat
updatestage
sprite(23).pPeriod = 0
sprite(23).pActive = FALSE
end doneCookin

```

SetTimerMode

```
property theTimer
```

```

on new me
  theTimer = 0
  put theTimer
  return me
end new

```

```

on SetTimer
  theTimer = 0
  put 0
end SetTimer

```

```

on addTimer me, addTime
  theTimer = theTimer * 10 + addTime
  if theTimer > 6000 then
    theTimer = theTimer mod 10000
  end if
  displayTime
  --sprite(23).pCurrentMinutes = theTimer / 100
  --sprite(23).pCurrentSeconds = theTimer mod 100
  --sprite(23).pPeriod = 60000 * sprite(23).pCurrentMinutes \
  --+ 1000 * sprite(23).pCurrentSeconds
  put theTimer
  --put sprite(23).pPeriod
end addTimer

```

```

on addMinute me
  if theTimer < 100 then
    theTimer = theTimer * 100
  end if
  if theTimer = 0 then
    theTimer = 60
  end if
  sprite(23).pCurrentMinutes = theTimer / 100
  sprite(23).pCurrentSeconds = theTimer mod 100
  sprite(23).pPeriod = 60000 * sprite(23).pCurrentMinutes \
  + 1000 * sprite(23).pCurrentSec onds
  displayTime
  put theTimer
  put sprite(23).pPeriod
end addTimer

```

```

on presetTimer me, setTime
  theTimer = setTime
  displayTime
  put theTimer
end presetTimer

```

```

on displayTime
  if theTimer/100 < 10 then
    if theTimer mod 100 < 10 then
      member("MicroTime").text = "0" & string(theTimer/100) & ":0" & string(theTimer mod 100)
    else
      member("MicroTime").text = "0" & string(theTimer/100) & ":" & string(theTimer mod 100)
    end if
  end if
else

```

```

if theTimer mod 100 < 10 then
  member("MicroTime").text = string(theTimer/100) & ":0" & string(theTimer mod 100)
else
  member("MicroTime").text = string(theTimer/100) & ":" & string(theTimer mod 100)
end if
end if
end displayTime

```

CookLevel

-- DESCRIPTION --

```

on getBehaviorDescription me
  return \
    "MULTISTATE TOGGLE BUTTON" & RETURN & RETURN & \
    "This behavior toggles the sprite it is attached to between two states: OFF and ON. " & \
    "In each state, the member of the sprite is set according to the position of the mouse (elsewhere, rollover,
mouseDown)." & RETURN & \
    "The behavior returns the current state of the button in response to a #ToggleButton_State call." & RETURN & \
  RETURN & \
    "RADIO BUTTON GROUP" & RETURN & \
    "To create a group of radio buttons, give each button in the same group the same ID. " & \
    "Switching one button in the group on will switch all others off." & RETURN & RETURN & \
    "PERMITTED MEMBER TYPES" & RETURN & \
    "Graphic" & RETURN & RETURN & \
    "PARAMETERS:" & RETURN & \
    "-- OFF state --" & RETURN & \
    "** Standard member (when mouse is elsewhere)" & RETURN & \
    "** Rollover member" & RETURN & \
    "** MouseDown member" & RETURN & \
    "-- ON state --" & RETURN & \
    "** Standard member" & RETURN & \
    "** Rollover member" & RETURN & \
    "** MouseDown member" & RETURN & \
    "-- COMMAND --" & RETURN & \
    "** Sent when the button is switched ON" & RETURN & \
    "** Sent when the button is switched OFF" & RETURN & \
    "-- Optional --" & RETURN & \
    "** Toggle group ID (to create a group of radio buttons)" & RETURN & RETURN & \
    "If members are placed consecutively in the cast in this order then default values can be used to create the button."
end getBehaviorDescription

on getBehaviorTooltip me
  return \
    "Create an ON/OFF button which reacts to rollovers and clicks. " & \
    "Use several such buttons together as a radio button group."
end getBehaviorTooltip

```

-- NOTES FOR DEVELOPERS --

```

--
-- This behavior communicates extensively with other sprites. Since there
-- may be up to 1000 sprites, it is important to use sendAllSprites with
-- prudence. Indeed, this behavior only uses sendAllSprites once, in the
-- Initialize handler.
--
-- ToggleButton_GroupRollCall
-- The purpose of the call is to identify all other members of the radio
-- button group. The behaviors that field the call add their object reference
-- to a list, ourGroupList. Each behavior stores a pointer to this list.
-- Subsequent communications are of the form:
--
-- call (#customMessage, ourGroupList, additionalParameters)
--
-- This limits the messaging process to only those sprites which need to know.

```

```

--
-- ToggleButton_MouseDownList
-- Lists can be used to make information available to other sprites without
-- sending any messages. All behaviors in a group also share a pointer to an
-- ourMouseDown list. Whenever the user clicks on one of the members, the
-- unique item in this list is set to TRUE. When the mouse is released, it
-- is set to FALSE. All buttons in the group know instantaneously if any
-- member of the group has been clicked. This means that if the mouse is
-- clicked on one button in the group, other group members will switch to
-- their mouseDown state as the mouse is dragged over them
--
-- ToggleButton_Rollover
-- When the mouse is released, the clickOn sprite asks each other group
-- member in turn if the mouse is currently over it. If none respond, then
-- ourMouseDown is set to [FALSE]. If any buttons respond, then the
-- mouseUp event sent to the topmost button will activate its Toggle handler.
--
-- ToggleButton_Off
-- When a button is toggled on, it instructs all others in the group to switch
-- themselves off.
--
-- ToggleButton_State, ToggleButton_ActiveButton
-- These two calls are not used in the behavior itself. they are included
-- to let you know the state of a particular button, or to know which button
-- in a given group is currently ON.
--
-- If you have only one button, then you can use the ToggleButton_State
-- call with no parameters. If there are several buttons, then you can either
-- send an empty list out to a given group, and receive a list of the state of
-- each button in reply.
--
-- ToggleButton_ActiveButton should be sent to a particular group. You
-- must include an empty property list in your call, and will receive in reply
-- a list of the form: [#sprite: <spriteNum>, #behavior: <object reference>]
--
--
-- HISTORY --
--
-- 11 September 1998, written for the D7 Behaviors Palette by James Newton
-- 5 January 2000: updated to D8 <km>

-- PROPERTIES --
global gOn
global gStopped
property spriteNum
property mySprite
-- author-defined parameters
property myOffMember
property myOffOverMember
property myOffDownMember
property myOffCommand
property myOnMember
property myOnOverMember
property myOnDownMember
property myOnCommand
-- internal properties
property theMouseWasUp
property myRollover
property myState -- TRUE | FALSE: max one button in group TRUE at one time
-- shared properties
property ourID -- string common to all buttons in a group
property ourGroupList -- list of behaviors in the group
property ourMouseDown -- list indicating if the clickOn is in the group

```

-- EVENT HANDLERS --

```
on beginSprite me
  Initialize me
end beginSprite
```

```
on prepareFrame me
  CheckForRollover me
end prepareFrame
```

```
on mouseDown me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    ClickOn me
  end if
end mouseDown
```

```
on mouseUp me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    if ourMouseDown[1] then Toggle me
  end if
end mouseUp
```

```
on mouseUpOutside me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    CheckGroupForClick me
  end if
end mouseUpOutside
```

-- CUSTOM HANDLERS --

```
on Initialize me -- sent by beginSprite
  mySprite = sprite(me.spriteNum)
  ourGroupList = []

  -- Insurance: properties are indeed #members
  myOffMember = member (myOffMember)
  myOffOverMember = member (myOffOverMember)
  myOffDownMember = member (myOffDownMember)
  myOnMember = member (myOnMember)
  myOnOverMember = member (myOnOverMember)
  myOnDownMember = member (myOnDownMember)

  sendAllSprites (#ToggleButton_GroupRollCall, ourID, ourGroupList)
  call (#ToggleButton_MouseDownList, ourGroupList, [FALSE])
end Initialize
```

```
on CheckForRollover me -- sent by prepareFrame
  mouseOverMe = (the rollover = me.spriteNum)
  if myRollover = mouseOverMe then
    if theMouseWasUp = the mouseUp then
      exit -- Nothing has changed
    end if
  end if

  else
    theMouseWasUp = the mouseUp
    if mouseOverMe then
      if the mouseUp then

```

```

-- Mouse was clicked elsewhere then dragged and released over button
case myState of
  TRUE: mySprite.member = myOnOverMember
  FALSE: mySprite.member = myOffOverMember
end case
end if
end if
else

set myRollover to mouseOverMe
if ourMouseDown[1] then
if myRollover then
  case myState of
    TRUE: mySprite.member = myOnDownMember
    FALSE: mySprite.member = myOffDownMember
  end case
else
  -- Indicate that mouseUpOutside will have no effect
  case myState of
    TRUE: mySprite.member = myOnMember
    FALSE: mySprite.member = myOffMember
  end case
end if
else
if not the mouseDown and myRollover then
  case myState of
    TRUE: mySprite.member = myOnOverMember
    FALSE: mySprite.member = myOffOverMember
  end case
else
  -- No reaction if mouse was clicked elsewhere and dragged to button
  case myState of
    TRUE: mySprite.member = myOnMember
    FALSE: mySprite.member = myOffMember
  end case
end if
end if
end CheckForRollover

```

```

on ClickOn me -- sent by mouseDown, CheckForRollover
ourMouseDown[1] = TRUE
case myState of
  TRUE: mySprite.member = myOnDownMember
  FALSE: mySprite.member = myOffDownMember
end case
end ClickOn

```

```

on Toggle me, whichState -- sent by mouseUp, ToggleButton_Off

```

```

if voidP (whichState) then
  myState = not myState
else
  myState = whichState
end if
ourMouseDown[1] = FALSE
theMouseWasUp = TRUE
case myState of
  TRUE:
    mySprite.member = myOnMember
    updateStage
  do myOnCommand
  FALSE:
    mySprite.member = myOffMember
    updateStage

```

```

        do myOffCommand
    end case
    if ourGroupList.count() then
        if myState then
            call (#ToggleButton_Off, ourGroupList, me)
        end if
    end if
end Toggle

on CheckGroupForClick me -- sent by mouseUpOutside
    groupRollover = call (#ToggleButton_Rollover, ourGroupList, [])
    if not groupRollover.count() then
        ourMouseDown[1] = FALSE
    end if
end Disactivate

-- PUBLIC METHODS (responses to #sendSprite, #sendAllSprites, #call) --

on ToggleButton_GroupRollCall me, groupID, groupList
    -- sent by each new button that joins the group
    if groupID = ourID then
        ourGroupList = groupList
        ourGroupList.append(me)
    end if
    return groupList
end ToggleButton_GroupRollCall

on ToggleButton_MouseDownList me, mouseDownList
    ourMouseDown = mouseDownList
end ToggleButton_MouseDownList

on ToggleButton_Rollover me, theList
    if the rollover = me.spriteNum then
        theList.append(me.spriteNum)
    end if
    return theList
end ToggleButton_Rollover

on ToggleButton_Off me, callingBehavior
    -- sent when the member of the group which is toggled ON
    if callingBehavior = me then exit
end ToggleButton_Off

Toggle me, FALSE
end ToggleButton_Off

on ToggleButton_State me, groupID, statesList
    if not voidP (groupID) then
        if groupID <> ourID then exit
        end if
    end if

    case ilk (statesList) of
        #void: return myState
        #list:
            statesList.append(myState)
        #propList:
            statesList.addProp(me.spriteNum, myState)
    end case
    return statesList
end ToggleButton_State

```

```

on ToggleButton_ActiveButton me, groupID, theList
  if not voidP (groupID) then
    if groupID <> ourID then exit
  end if

  if not listP (theList) then
    theList = [:]
  end if
  if not theList.count() and myState then
    theList.addProp (#sprite, me.spriteNum)
    theList.addProp (#behavior, me)
  end if
  return theList
end ToggleButton_ActiveButton

```

```

on isOKToAttach (me, aSpriteType, aSpriteNum)

```

```

  tIsOK = 0
  if aSpriteType = #graphic then
    tIsOK = 1
  end if

```

```

  return(tIsOK)
end on

```

```

-- AUTHOR-DEFINED PARAMETERS --

```

```

on getPropertyDescriptionList me

```

```

  theMember = sprite(the currentSpriteNum).member
  theMemberNum = theMember.number

```

```

  return \
  [ \
  #myOffMember: \
  [ \
  #comment: "-OFF STATE- Standard member:", \
  #format: #graphic, \
  #default: theMember \
  ], \
  #myOffOverMember: \
  [ \
  #comment: "Rollover member", \
  #format: #graphic, \
  #default: member (theMemberNum + 1) \
  ], \
  #myOffDownMember: \
  [ \
  #comment: "MouseDown member", \
  #format: #graphic, \
  #default: member (theMemberNum + 2) \
  ], \
  #myOnMember: \
  [ \
  #comment: "-ON STATE- Standard member", \
  #format: #graphic, \
  #default: member (theMemberNum + 3) \
  ], \
  #myOnOverMember: \
  [ \
  #comment: "Rollover member", \
  #format: #graphic, \
  #default: member (theMemberNum + 4) \
  ], \
  #myOnDownMember: \

```

```

[ \
#comment: "MouseDown member", \
#format: #graphic, \
#default: member (theMemberNum + 5) \
], \
#myOnCommand: \
[ \
#comment: "-COMMAND SENT- when switched ON", \
#format: #string, \
#default: "sendAllSprites #Toggle_On, the currentSpriteNum" \
], \
#myOffCommand: \
[ \
#comment: "when switched OFF", \
#format: #string, \
#default: "sendAllSprites #Toggle_Off, the currentSpriteNum" \
], \
#ourID: \
[ \
#comment: "ID string for the group", \
#format: #string, \
#default: EMPTY \
] \
] \
end getPropertyDescriptionList

```

Countdown Timer

```

-- Timer - Countdown
-- Timer can be set to any value up to 24 days and count down to 0.
-- Behavior can then send a message.
-- v1 - 23 September 1998 by Darrel Plant
-- v2 - 10 January 2000 by Chris Walcott. Added isOKToAttach.
-- Removed unnecessary error checking.

```

```

on getBehaviorDescription me
return \
"TIMER - COUNTDOWN" & RETURN & RETURN & \
"Placed on a text or field sprite, will count backward to 0 from the time set in the
Property dialog. " & \
"The clock can count down a period of up to 24 days. " & \
"Values can be displayed in with numbers representing days, hours, minutes, seconds,
and hundredths. " & \
"When the timer runs out, a global handler can be called or a message can be sent to
all sprites. " & \
"Uses the font selected for the text or field cast member. " & \
"The current time may be broadcast to other sprites; if that option is checked, an
mCountdown message will be sent, with a string containing the value of the countdown
timer as a parameter. " & \
"The countdown can start in the first frame the sprite appears, or it can wait for an
mCountdownStart message to be sent.PARAMETERS:" & RETURN & \
" - Initializing event for countdown" & RETURN & \
" - Largest displayed time unit" & RETURN & \
" - Smallest displayed time unit" & RETURN & \
" - Initiating event" & RETURN & \
" - Days to count down" & RETURN & \
" - Hours to count down" & RETURN & \
" - Minutes to count down" & RETURN & \
" - Seconds to count down" & RETURN & \
" - Message to send" & RETURN & \
" - Broadcast time to other sprites"
end getBehaviorDescription

on getBehaviorTooltip me
return \
"[Advanced]" & RETURN & RETURN & \
"Use this to display a digital clock with days, hours, minutes, seconds, and
hundredths of a second, starting at the time you set and counting down to 0. " & \

```

```

    "Drop it on the text or field that will be used to display the timer." & RETURN &
RETURN & \
    "The time can send a message when it reaches 0 and broadcast the remaining time to
other sprites."
end getBehaviorTooltip

```

```
-- PROPERTIES --
```

```

property pSprite          -- sprite channel number
property pMember          -- cast member
property pInitialize      -- when to start countdown
property pLargeUnit       -- largest time unit to display
property pLargeIndex      -- index value referencing pLargeUnit
property pSmallUnit       -- smallest time unit to display
property pSmallIndex      -- index value referencing pSmallUnit
property pStartDays       -- days to count down
property pStartHours      -- hours to count down
property pStartMinutes    -- minutes to count down
property pStartSeconds    -- seconds to count down
property pStartMillis     -- reference start time
property pCurrentDays     -- days to display
property pCurrentHours    -- hours to display
property pCurrentMinutes  -- minutes to display
property pCurrentSeconds  -- seconds to display
property pCurrentMillis   -- milliseconds to display
property pMessage        -- message to send when timer reaches 0
property pMessageDest     -- where to send message
property pBroadcast       -- flag to send time to other sprites
property pRollover        -- set if millisecond counter has maxed out
property pActive          -- counter is counting
property pTimeUnits       -- list of time unit values
property pPeriod          -- total period of timer in milliseconds
property vElapsed
property vMillis
global gState
global gClockOn

```

```
-- EVENT HANDLERS
```

```

on resolve (prop)
  case prop of
    pLargeUnit:
      choicesList = ["Days", "Hours", "Minutes", "Seconds"]
      lookup = [#days, #hours, #minutes, #seconds]
    pSmallUnit:
      choicesList = ["Hours", "Minutes", "Seconds", "Hundredths"]
      lookup = [#hours, #minutes, #seconds, #hundredths]
    otherwise: nothing
  end case
  return lookup[findPos(choicesList, prop)]
end resolve

```

```

on beginSprite me
  pLargeUnit = resolve(pLargeUnit)
  pSmallUnit = resolve(pSmallUnit)
  mInitialize me
end beginSprite

```

```

on prepareFrame me
  mDisplayTime me
end prepareFrame

```

```
-- CUSTOM HANDLERS --
```

```

on mDisplayTime me
  -- updates counter; called from prepareFrame and mInitialize
  -- counter is updated only when active
  if pActive then
    -- determine current value of millisecond counter
    vMillis = the milliseconds
    if vMillis < pStartMillis then

```

```

-- indicates that the millisecond counter has reached
-- its maximum value and has reset to 0
pRollover = TRUE
end if
-- determines number of milliseconds since countdown started
put vMillis
put pStartMillis
vElapsed = vMillis - pStartMillis
put vElapsed
if pRollover then
-- if millisecond counter has reset since countdown started,
-- vElapsed will be negative; adding maxInteger obtains correct
-- number of milliseconds since countdown started
vElapsed = vElapsed + the maxinteger
end if
-- subtracts elapsed time from the calculated total number of
-- milliseconds allotted for countdown
vElapsed = pPeriod - vElapsed
if vElapsed < 0 then
-- countdown has ended
-- because countdown may not be evaluated exactly at time = 0,
-- this accounts for that possibility
--vElapsed = 0
-- deactivate the countdown timer
pActive = FALSE
if pMessage <> "none" then
-- send a message when countdown has reached 0
if pMessageDest = "movie scripts" then
-- call movie handler or evaluate command
do pMessage
else
-- send message to all sprites
sendAllSprites (symbol (pMessage))
end if
end if
end if
-- find current values for display
-- all time units are evaluated, regardless of whether they're
-- actually shown
pCurrentDays = vElapsed / 86400000
vElapsed = vElapsed - (pCurrentDays * 86400000)
pCurrentHours = vElapsed / 3600000
vElapsed = vElapsed - (pCurrentHours * 3600000)
pCurrentMinutes = vElapsed / 60000
vElapsed = vElapsed - (pCurrentMinutes * 60000)
pCurrentSeconds = vElapsed / 1000
vElapsed = vElapsed - (pCurrentSeconds * 1000)
pCurrentMillis = vElapsed
-- initialize display string
vDisplay = ""
repeat with i = 1 to 5
-- build display string
if (i >= pSmallIndex) and (i <= pLargeIndex) then
-- add items to display string only for selected time units
if vDisplay <> "" then
-- add separators between time unit values
case i of
1: -- hundredths of a second don't have a separator
nothing
2: -- decimal point shows between seconds and hundredths
vDisplay = "." & vDisplay
otherwise
-- other units use colon for separator (in US, anyway)
vDisplay = ":" & vDisplay
end case
end if
end if
case i of
-- add time units to display string
-- all time unit values are two digits (0..99, 0..59, or 0..24)
-- depending on the unit; to get leading 0 values, add 100 to
-- the number (100 + 1 = 101), convert to a string ("101"),

```

```

        -- and derive the two characters on the right ("01")
    1: vDisplay = mRightStr ((pCurrentMillis / 10 + 100), 2)
    2: vDisplay = mRightStr ((pCurrentSeconds + 100), 2) & vDisplay
    3: vDisplay = mRightStr ((pCurrentMinutes + 100), 2) & vDisplay
    4: vDisplay = mRightStr ((pCurrentHours + 100), 2) & vDisplay
    5: vDisplay = mRightStr ((pCurrentDays + 100), 2) & vDisplay
    end case
    end if
end repeat
-- put display string into sprite's member
pMember.text = vDisplay
if pBroadcast then
    -- send time to other sprites if flag is set
    sendAllSprites (#mCountdown, vDisplay)
end if
else -- display time
    if gState = 0 then
        pMember.text = chars(the time,1,5) & " " & chars(the time,4,5)
    end if
end if
end mDisplayTime

on mInitialize me
    -- initializes values for countdown timer
    -- derive sprite object reference
    pSprite = sprite (me.spriteNum)
    -- find sprite's cast member reference
    if me.spriteNum < 0 then
        -- script is attached to wrong channel
        return mErrorAlert (me, #channelProb)
    end if
    pMember = pSprite.member
    -- initialize time units list
    -- if not mPermittedMemberTypes ().getPos (pMember.type) then
    --     -- sprite member is not of the correct type
    --     return mErrorAlert (me, #wrongType, me.spriteNum)
    -- end if
    mInitTimeUnits me
    -- set up parameters for largest and smallest displayed time units
    pSmallIndex = getPos (pTimeUnits, pSmallUnit)
    pLargeIndex = getPos (pTimeUnits, pLargeUnit)
    pSmallIndex = min (pSmallIndex, pLargeIndex)
    -- initialize countdown timer
    pStartMillis = the milliseconds
    -- set flag for millisecond timer rollover
    pRollover = FALSE
    -- set activity level of countdown timer; countdown may begin
    -- when the sprite is instantiated or wait for a message to
    -- be sent to the sprite
    if pInitialize = "first frame of sprite" then
        pActive = TRUE
    else
        pActive = FALSE
    end if
    -- set current time unit values
    pCurrentDays = pStartDays
    pCurrentHours = pStartHours
    pCurrentMinutes = pStartMinutes
    pCurrentSeconds = pStartSeconds
    pCurrentMillis = 0
    -- find total number of milliseconds to count down
    pPeriod = 86400000 * pStartDays + 3600000 * pStartHours \
        + 60000 * pStartMinutes + 1000 * pStartSeconds
    -- initialize time display
    mDisplayTime me
end mInitialize

```

```

on mInitTimeUnits me
  -- set up list of time units in order of smallest to largest
  pTimeUnits = [#hundredths, #seconds, #minutes, #hours, #days]
end mInitTimeUnits

on mRightStr vString, vCount
  -- return the rightmost characters of a string
  -- in a more uncontrolled environment, error checking for
  -- short strings should be implemented
  vString = string(vString)
  vLength = vString.length
  return vString.char[(vLength - 1)..vLength]
end mRightStr

on mErrorAlert me, vError, vData
  -- based on James Newton's error checking procedure
  -- determine name of behavior
  vBehaviorName = string(me)
  delete word 1 of vBehaviorName
  delete the last word of vBehaviorName
  delete the last word of vBehaviorName
  -- convert supporting data
  case vData.ilk of
    #void: vData = "<void>"
    #symbol: vData = "#" & vData
  end case
  case vError of
    -- deal with individual error types
    #channelProb:
      alert "Countdown Timer behavior is attached to a non-sprite channel"
    -- #wrongType:
    --   alert "Sprite" && vData && ": Countdown Timer sprite MUST be one of the
following member types:" \
    --   & RETURN & mPermittedMemberTypes ()
  end case
end mErrorAlert

-- PUBLIC METHODS --

on mCountdownStart me
  -- activates dormant countdown timer
  pActive = TRUE
  -- sets base time for countdown start
  pStartMillis = the milliseconds
end mCountdownStart

-- AUTHOR-DEFINED PARAMETERS --

on isOKToAttach (me, aSpriteType, aSpriteNum)

  case aSpriteType of
    #graphic:
      return getPos([#field, #text], sprite(aSpriteNum).member.type) <> 0
    #script:
      return FALSE
  end case

end

on getPropertyDescriptionList me
  vPDLList = []
  setaProp vPDLList, #pInitialize, [#comment: "Initializing event", \
  #format: #string, #default: "first frame of sprite", \
  #range: ["first frame of sprite", "mCountdownStart message"]]
  setaProp vPDLList, #pLargeUnit, [#comment: "Largest time unit", \
  #format: #string, #default: "Seconds", #range: ["Days", "Hours", "Minutes",
"Seconds"]]
  setaProp vPDLList, #pSmallUnit, [#comment: "Smallest time unit", \

```

```

    #format: #string, #default: "Seconds", #range: ["Hours", "Minutes", "Seconds",
"Hundredths"]
    setaProp vPDList, #pStartDays, [#comment: "Days to count down", \
    #format: #integer, #default: 0, #range: [#min:0, #max: 23]]
    setaProp vPDList, #pStartHours, [#comment: "Hours to count down", \
    #format: #integer, #default: 0, #range: [#min: 0, #max: 23]]
    setaProp vPDList, #pStartMinutes, [#comment: "Minutes to count down", \
    #format: #integer, #default: 0, #range: [#min: 0, #max: 59]]
    setaProp vPDList, #pStartSeconds, [#comment: "Seconds to count down", \
    #format: #integer, #default: 0, #range: [#min: 0, #max: 59]]
    setaProp vPDList, #pMessage, [#comment: "Message to send after countdown", \
    #format: #string, #default: "none"]
    setaProp vPDList, #pMessageDest, [#comment: "Where to send message", \
    #format: #string, #default: "all sprites", #range: ["all sprites", "movie scripts"]]
    setaProp vPDList, #pBroadcast, [#comment: "Broadcast countdown time?", \
    #format: #boolean, #default: FALSE]
    return vPDList
end getPropertyDescriptionList

--on mPermittedMemberTypes
-- -- returns list of permitted media types that
-- -- may be used for countdown timer
-- return [#text, #field]
--end mPermittedMemberTypes

```

Display Text

-- DESCRIPTION --

```

on getBehaviorDescription me
    return \
        "DISPLAY TEXT" & RETURN & RETURN & \
        "This behavior allows you to display a given string in a field or text member. " & \
        "Use it with the Tooltip and Hypertext - Display Status behaviors which need a field or text member in which to display
their information. " & \
        "Or create your own custom Lingo to display runtime information, such as the position of the mouse." & RETURN &
RETURN & \
        "This behavior waits for Lingo commands to tell it what to do. " & \
        "It is not active by itself." & RETURN & RETURN & \
        "You can choose between two display types: tooltip and status bar." & RETURN & RETURN & \
        "The TOOLTIP type of display will make the field or text member resize itself to fit the text, and disappear when it is
empty. " & \
        "You can set the tooltip type display to appear at any position on the stage, such as under the cursor. " & \
        "If no position is sent to the sprite, it will appear at the top left corner of the Stage. " & \
        "See the Tooltip behavior for more details." & RETURN & RETURN & \
        "If you wish to display several lines of text, you must use RETURN characters to define the line breaks. " & \
        "An empty tooltip sprite will move off-stage to hide. " & \
        "It is recommended that you place it off-stage before it is used, in case it causes a brief flash on the screen." &
RETURN & RETURN & \
        "The STATUS BAR type of display will appear on Stage at all times. " & \
        "It will not resize or change position. " & \
        "Any positional information sent to this sprite will be ignored if it is set to act as a status bar. " & \
        "If the text is too long to appear in the member of the current sprite, a scrollbar will appear. " & \
        "You do not need to divide the text with RETURN characters. " & \
        "If you think that a scrollbar may be necessary, make sure that the field or text member is sufficiently tall for the scroll
arrows to operate correctly." & RETURN & RETURN & \
        "Set the font size and other characteristics of the field or text member to customize the appearance of the message." &
RETURN & RETURN & \
        "Be sure to give the field or text member a name. " & \
        "It may be emptied by this behavior. " & \
        "Director automatically erases nameless empty members." & RETURN & RETURN & \
        "PERMITTED MEMBER TYPES:" & RETURN & \
        "field and text" & RETURN & RETURN & \
        "PARAMETERS:" & RETURN & \
        "** Display type:" & RETURN & \
        " - Tooltip (appears near the cursor on rollover)" & RETURN & \
        " - Status bar (appears in a fixed position at all times)" & RETURN & RETURN & \
        "PUBLIC METHODS:" & RETURN & \

```

```

    "** Set the text to display (and the position of the sprite)" & RETURN & RETURN & \
    "ASSOCIATED BEHAVIORS:" & RETURN & \
    "** Tooltip" & RETURN & \
    "** Source Status" & RETURN & \
    "** Hypertext - Display Status"
end getBehaviorDescription

on getBehaviorTooltip me
return \
    "Use with field or text members." & RETURN & RETURN & \
    "Waits for a message from another behavior or custom handler to display a character string." & \
    "This behavior is intended to be used with the Tooltip and Hypertext - Display Status behaviors to create a status bar or
a tooltip under the cursor."
end getBehaviorTooltip

```

-- NOTES FOR DEVELOPERS --

-- COMMUNICATING WITH THE SPRITE

-- To set the text of the current sprite's member, use a line similar to one of
-- the following:

```

-- sendAllSprites (#DisplayText_SetText, theStringToDisplay)
--
-- sendSprite (<spriteNumber>, #DisplayText_SetText, theStringToDisplay)
--
-- call (#DisplayText_SetText, displayBehavior, theStringToDisplay)
--

```

-- It is fastest to call the behavior directly. Use code similar to the
-- following lines in any script that needs to communicate with this behavior
-- often:

```

--
-- property displayBehavior
--
-- ...
-- displayBehavior = []
-- sendAllSprites (#DisplayText_Enroll, displayBehavior)
--
-- ...
--

```

-- If your _Enroll call fails for some reason, displayBehavior will be an
-- empty list. Calling an empty list will not produce an error (calling an
-- invalid behavior reference will). It would, however, be wise to provide
-- yourself with an authortime alert so that you can correct such a problem
-- if necessary:

```

--
-- if displayBehavior.count() = 0 then
--   if the runMode = "Author" then
--     alert "No 'Display Text' behavior found in frame "the frame
--     end if
--   end if
--

```

-- Writing this sort of "defensive" code should ensure that any bugs that make
-- it through to the finished product are relatively harmless.

-- ADJUSTING THE WIDTH OF THE SPRITE

-- The BestRect function is called if you choose the Tooltip display type. It
-- modifies the width of the sprite to suit the longest line/paragraph of text \
-- (field lines and text paragraphs are delimited by the RETURN character).

-- This function uses the charPosToLoc() function to determine the length of
-- each line. CharPosToLoc returns the position of the bottom left corner of
-- the tested character: if the character tested is the final RETURN of a line,
-- then the value returned is equivalent to the bottom right of the last
-- visible character in the line.

-- I initially set the width of the member to an extravagantly large value, to
-- ensure that no line-wrapping should occur.

-- HISTORY --

-- 1 october 1998: Written for the D7 Behaviors Palette by James Newton
-- 28 October 1998: Descriptions improved. getPDL simplified, GetTopLeft() added
-- 7 January 2000: Added isOKToAttach and substituteStrings event handlers
-- and removed redundant error checking code - Karl Miller

-- PROPERTIES --

```
property spriteNum
-- error checking
property getPDLError
-- author-defined parameters
property myDisplayType
-- internal properties
property mySprite
property myMember
property myWidthAdjust
property myHeightAdjust
property myOffStageLoc
```

-- EVENT HANDLERS --

```
on beginSprite me
  myDisplayType = resolve(myDisplayType)
  Initialize me
end beginSprite
```

```
on endSprite me
  mySprite.visible = TRUE
end endSprite
```

-- CUSTOM HANDLERS --

```
on resolve(prop)
  case prop of
  myDisplayType:
    choicesList = ["status bar (fixed size and position)", \
                  "tooltip (dynamic size and position)"]
    lookup = [#statusBar, #tooltip]
  end case
  return lookup[findPos(choicesList, prop)]
end resolve
```

```
on Initialize me -- sent by beginSprite
  mySprite = sprite(me.spriteNum)
  myMember = mySprite.member

  if myMember.type = #field then
    myWidthAdjust = (myMember.margin + myMember.border) * 2
    myHeightAdjust = myMember.margin + myMember.border * 2
    -- no gremlins here:)
  else
    myWidthAdjust = 0
    myHeightAdjust = 0
  end if
  myMember.text = EMPTY

  if myDisplayType = #tooltip then
    myMember.boxType = #fixed
    myOffStageLoc = point (999, 999)
```

```

    mySprite.loc = myOffStageLoc
end if
end Initialize

```

```

on BestRect me, theString -- sent by DisplayText_SetText
-- Sets the rect of myMember to fit snugly round theString it displays

```

```

myMember.rect = rect (0, 0, 8000, 0)
myMember.text = theString -- Needed to update myMember.rect
bestRect = myMember.rect
theLine = the number of lines of theString
theWidth = 0
checkedChars = 0
-- Determine the length of the longest line
repeat while theLine
endOfLine = offset (RETURN, theString)
if not endOfLine then
-- Only one line remaining
endOfLine = (the number of chars of theString) + 1
myMember.text = myMember.text & RETURN
end if
checkedChars = checkedChars + endOfLine
endPoint = charPosToLoc (myMember, checkedChars)
lineWidth = endPoint[1]
if lineWidth > theWidth then
theWidth = lineWidth
end if
delete char 1 to endOfLine of theString
theLine = theLine - 1
end repeat
-- Determine the height of the text
lastChar = myMember.char.count
lastCharLoc = charPosToLoc (myMember, lastChar)
theHeight = lastCharLoc[2]
--
bestRect[3] = theWidth + 1
bestRect[4] = theHeight + 1
return bestRect
end BestRect

```

```

on GetTopLeft me, theLoc, theAlignment, memberRect
case theAlignment of
#bottomCenter: return theLoc - [memberRect.width / 2, memberRect.height]
#bottomRight: return theLoc - [memberRect.width, memberRect.height]
#bottomLeft: return theLoc - [0, memberRect.height]
#center: return theLoc - [memberRect.width / 2, memberRect.height / 2]
#topCenter: return theLoc - [memberRect.width / 2, 0]
#topRight: return theLoc - [memberRect.width, 0]
otherwise-- treat as #topLeft
return theLoc
end case
end GetTopLeft

```

```

-- INTER-SPRITE COMMUNICATION (response to #sendSprite, #sendAllSprites) --

```

```

on DisplayText_Enroll me, enrollList
-- sent by objects which need to call this specific behavior
if ilk (enrollList) <> #list then return me
if not enrollList.count() then
enrollList.append(me)
else
-- the calling behavior has already found a candidate
end if
return enrollList
end DisplayText_Enroll

```

```

on DisplayText_SetText me, theString, theLoc, theAlignment
-- called by other objects
-- Sets the text of myMember to theString and shows mySprite
-- as near to theLoc as possible (if myDisplayType is #tooltip)
--
-- theAlignment can take any of the following values:
-- #bottomCenter|#bottomRight|#bottomLeft|#center|#topCenter|#topRight|#topLeft
-- This determines which point of the current sprite is to appear at theLoc

-- Error check
if not stringP (theString) then
  ErrorAlert (me, #invalidString, theString)
  theString = string (theString)
else
  case ilk (theLoc) of
    #void, #point: -- nothing
    otherwise
      ErrorAlert (me, #invalidPoint, theLoc)
      theLoc = point (0, 0)
  end case
end if
-- End of error check

if theString = EMPTY and myDisplayType = #tooltip then
  mySprite.loc = myOffStageLoc
else

  myMember.text = theString
  if myDisplayType = #tooltip then
    memberRect = BestRect (me, theString)
    myMember.rect = memberRect
  else
    memberRect = myMember.rect
  end if
  memberRect = memberRect + [0, 0, myWidthAdjust, myHeightAdjust]
  if myDisplayType = #tooltip then
    if ilk (theLoc) <> #point then
      theLoc = point (0, 0)
    end if
    theLoc = GetTopLeft (me, theLoc, theAlignment, memberRect)
    -- Ensure sprite is fully visible on stage
    stageWidth = (the activeWindow).rect.right - (the activeWindow).rect.left
    stageHeight = (the activeWindow).rect.bottom - (the activeWindow).rect.top
    maxH = stageWidth - memberRect.width
    maxV = stageHeight - memberRect.height
    theLoc[1] = max (0, min (theLoc[1], maxH))
    theLoc[2] = max (0, min (theLoc[2], maxV))

    theLoc = theLoc + myMember.regPoint
    mySprite.loc = theLoc
  else
    lastChar = theString.char.count
    textHeight = charPosToLoc (myMember, lastChar)[2]
    if textHeight > mySprite.height then
      myMember.boxType = #scroll
    else
      myMember.boxType = #fixed
    end if
  end if
end if
end DisplayText_SetText

on DisplayText_GetReference me
  return me
end DisplayText_GetReference

```

```
-- ERROR CHECKING --
```

```
on ErrorAlert me, theError, data
  -- Determine the behavior's name
  behaviorName = string (me)
  delete word 1 of behaviorName
  delete the last word of behaviorName
  delete the last word of behaviorName

  case theError of
    #invalidString:
      if the runMode = "Author" then
        message = substituteStrings(me, \
"BEHAVIOR ERROR: Frame ^0, Sprite ^1" & RETURN & \
"Behavior ^2" & RETURN & RETURN & \
"The DisplayText_SetText handler could not treat the following as a string:" & RETURN & RETURN & \
"^3", \
["^0": the frame, "^1": me.spriteNum, "^2": behaviorName, "^3": data])
        alert message
      end if
    #invalidPoint:
      if the runMode = "Author" then
        message = substituteStrings(me, \
"BEHAVIOR ERROR: Frame ^0, Sprite ^1" & RETURN & \
"Behavior ^2" & RETURN & RETURN & \
"The DisplayText_SetText handler could not treat the following as a point:" & RETURN & RETURN & \
"^3", \
["^0": the frame, "^1": me.spriteNum, "^2": behaviorName, "^3": data])
        end if
      end case
end ErrorAlert
```

```
on substituteStrings(me, parentString, childStringList) -----
  --
  -- * Modifies parentString so that the strings which appear as
  -- properties in childStringList are replaced by the values
  -- associated with those properties.
  --
  -- <childStringList> has the format ["^1": "replacement string"]
  -----
```

```
i = childStringList.count()
repeat while i
  tempString = ""
  dummyString = childStringList.getPropA(i)
  replacement = childStringList[i]
  lengthAdjust = dummyString.char.count - 1
  repeat while TRUE
    position = offset(dummyString, parentString)
    if not position then
      parentString = tempString&parentString
      exit repeat
    else
      if position <> 1 then
        tempString = tempString&parentString.char[1..position - 1]
      end if
      tempString = tempString&replacement
      delete parentString.char[1..position + lengthAdjust]
    end if
  end repeat
  i = i - 1
end repeat

return parentString
end substituteStrings
```

-- AUTHOR-DEFINED PARAMETERS --

```
on isOKToAttach (me, aSpriteType, aSpriteNum)
  case aSpriteType of
    #graphic:
      return getPos([#field, #text], sprite(aSpriteNum).member.type) <> 0
    #script:
      return FALSE
  end case
end isOKToAttach
```

on getPropertyDescriptionList me

```
  if not the currentSpriteNum then exit

  return \
  [ \
    #myDisplayType: \
    [ \
      #comment: "Display Text sprite behaves as a", \
      #format: #string, \
      #default: "status bar (fixed size and position)", \
      #range: ["status bar (fixed size and position)", \
        "tooltip (dynamic size and position)"] \
    ] \
  ] \
end getPropertyDescriptionList
```

EZ

-- DESCRIPTION --

on getBehaviorDescription me

```
  return \
    "PUSH BUTTON" & RETURN & RETURN & \
    "This behavior sets the member of a sprite depending on the state of the mouse (elsewhere, rollover, mouseDown, mouseUp)." & RETURN & RETURN & \
    "This creates a button which can either initiate actions in other sprites, or provide visual feedback for other behaviors attached to the same sprite." & RETURN & RETURN & \
    "The behavior can be enabled or disabled, using a #PushButton_ToggleActive call to the behavior or the sprite." & RETURN & RETURN & \
    "Two messaging systems are provided:" & RETURN & \
    "1) A custom message can sent whenever the Push Button behavior is activated. " & \
    "This message can be sent to a Movie Script handler, all sprites, or the actorList. " & \
    "The message can also be suppressed." & RETURN & RETURN & \
    "2) Objects can 'subscribe' to the behavior in order to receive PushButton_Activated, _Enabled and _Disabled messages." & \
    "A two-way messaging system allows for cleaning up object references before an object is destroyed." & RETURN & RETURN & \
    "The behavior can be set to consider that all sprites in higher channels either block or let through all mouse events. " & \
    "If mouse events are allowed to pass, you can place blended sprites above the button to change its color. " & \
    "If mouse events are blocked, such translucent sprites provide an alternative method for disabling the button." & RETURN & RETURN & \
    "PERMITTED MEMBER TYPES" & RETURN & \
    "[#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, #shape, #vectorShape]" & RETURN & RETURN & \
    "PARAMETERS:" & RETURN & \
    "** Standard member (when mouse is elsewhere)" & RETURN & \
    "** Rollover member" & RETURN & \
    "** MouseDown member" & RETURN & \
    "** Disabled member" & RETURN & RETURN & \
    "Optional parameters:" & RETURN & \
    "** MouseDown sound" & RETURN & \
    "** MouseUp sound" & RETURN & \
    "If members are placed consecutively in the cast in this order then default values can be used to create the button." & RETURN & RETURN & \
```

```

    "** Do sprites above the button allow mouse events through?" & RETURN & \
    "** Type of message sent on mouseUp: do | sendAllSprites | call the actorList | no action" & RETURN & \
    "** Custom Message sent on mouseUp" & RETURN & RETURN & \
    "NOTES:" & RETURN & \
    "If you use 'do', be sure that you have a handler in a Movie Script that corresponds to the message sent." & RETURN &
RETURN & \
    "If you indicate 'no action' then this behavior will simply deal with the different states of the button." & \
    "You can still execute an action on mouseUp by one of two means:" & RETURN & \
    "1) Add a behavior with a mouseUp handler to the same sprite (for example, the 'Jump Back Button' behavior" &
RETURN & \
    "2) Subscribe an object to the current behavior." & \
    "The object will then receive your Custom Message directly."
end getBehaviorDescription

```

```

on getBehaviorTooltip me
return \
    "Use with graphic members." & RETURN & RETURN & \
    "Swaps the member of the sprite according to the state of the mouse." & \
    "You can use this dynamic button behavior to play a brief sound on mouseDown and/or mouseUp, send out a custom
message of your choice, and trigger actions of other sprites." & \
    "You can also use it to provide visual feedback for other behaviors on the same sprite (for example, Jump to Marker
Button)." & \
    "This behavior can also interact with custom objects."
end getBehaviorTooltip

```

-- NOTES FOR DEVELOPERS --

-- A BUTTON BEHAVIOR WITH NO MOUSE EVENTS

-- This behavior makes a sprite react when it is clicked on. You might expect
-- to find mouseUp and mouseDown events... but there aren't any. There are no
-- mouseEnter or mouseLeave handlers either, and yet the behavior reacts to
-- rollover.
--
-- The reason for this is that Director is in two minds about intervening
-- sprites. Suppose there is a sprite between the mouse and the button graphic
-- that this behavior is attached to. Such an intervening sprite is opaque
-- for mouseEnter and mouseLeave: the button graphic beneath never receives
-- these events. However, the intervening sprite may be transparent for
-- mouseUp and mouseDown events: if it has no mouseUp/mouseDown handlers
-- attached (through a behavior or Cast Member script), then it lets the button
-- graphic beneath react to these events.
--
-- Relying on mouse events may lead to an ambiguous situation where the button
-- does not indicate that it is active on rollover yet it reacts when it is
-- clicked on.

-- ROLL-YOUR-OWN MOUSE EVENTS

-- To avoid this ambiguity, I let you choose in the Behavior Parameters dialog
-- whether the button can see through intervening sprites (in which case
-- it reacts both to rollover and mouseClicks) or whether it treats intervening
-- sprites as completely opaque (in which case it doesn't react at all).
-- This information is stored in myXRyFlag parameter. In order to provide
-- this choice I had to abandon the mouseEnter and mouseLeave events.

-- Instead, I test the state of where the mouse is and whether it is up or
-- down, on each prepareFrame. If you are using X-ray mode then I test for
-- rollover (spriteNum). If not, I test for (the mouseMember = myMember).
-- This ignores the transparent parts of intervening sprites with matte ink.

-- Both the prepareFrame and the exitFrame handlers are suitable places to do
-- this. I chose to use prepareFrame to run a CheckRollover handler. It is
-- important that actions only take place when the rollover or mouseDown states
-- change. The property myRollover remembers rollover from one frame to the
-- next, and the property myMouseDown remembers the mouseDown state. Together
-- these properties determine whether the member to display should be
-- myRolloverMember, myMouseDownMember or myStandardMember.

```

-- The theMouseWasUp property has a subtle role. It ensures that the button
-- is not activated if the mouse is clicked elsewhere, then dragged
-- and released over the button. A good part of the CheckForRollover handler
-- deals with this one rare case. The Activate handler sets theMouseWasUp
-- to FALSE. This means that the button will appear in its rollover state
-- after a click. If you prefer to show the standard state after a click,
-- then set theMouseWasUp to TRUE in the Activate handler.

-- TRIGGERING EVENTS
-- A button is not very useful if it can't be used to trigger events
-- elsewhere. I have built in four possibilities in two flavors:
-- 1) sending a custom message to
--   * a Movie Script handler (using a "do" command)
--   * all other sprites (using a "sendAllSprites" command)
--   * objects on the actorList (using a "call ... " & \
--     "the actorList" command)
-- 2) calling a list of subscribed objects:
--   call (#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)
--
-- Using "do"
-- If you use the "do" command, you must ensure that there is a handler in a
-- Movie Script which corresponds to the CustomMessage that you send.
-- If you do not do this, Director will indicate an error
--
-- Using "sendAllSprites"
-- Any sprites that are to be affected when the button is activated must have
-- an 'on YourCustomMessage me, callingBehaviorRef, callingSpriteNum' handler
-- in one of their behaviors.

-- #YourCustomMessage has to be a single word otherwise it can't be
-- converted to a symbol. The getPropertyDescriptionList handler
-- could specify #symbol format. This would clip the author's input at the
-- end of the first word, without warning. I prefer to alert authors that their
-- chosen message is not valid, so I deal with the #symbol conversion myself.
--
-- Calling objects on the actorList
-- The actorList is a convenient place for storing objects without using global
-- variables. You can send a message to all objects on the actorList without
-- having to identify the separate objects. All objects with an appropriate
-- handler will respond to the message. If no objects contain an appropriate
-- handler, the message is simply ignored.

-- The actorList is "local" to each window. Each MIAW has a separate actorList.
-- This means that a button on the stage cannot interact directly with an
-- object in the actorList of a MIAW.
--
-- Calling a list of subscribing behaviors or objects
-- You can also inform your other behaviors that the Push Button behavior has
-- been instantiated. To do this, add the following handler to any behavior
-- that needs to receive PushButton_Activated messages. You can use the
-- parameters 'theSprite' and 'message' to ensure that only the right
-- Push Button is treated.
--
--   property myButtonRef, myExpectedMessage, myButtonSprite
--
--   on PushButton_OpenForBusiness me, theList, buttonRef, theSprite, message
--   if not voidP (message) then
--     if message <> myExpectedMessage then exit -- Sent by the wrong button
--   end if
--   -- Store data concerning the button (this is optional)
--   myButtonRef = buttonRef-- Object reference for the button behavior
--   myButtonSprite = theSprite -- Sprite number of button
--   -- Tell the button behavior to call this object
--   theList.append(me)
--   return theList
-- end PushButton_OpenForBusiness
--
-- If the Push Button behavior is instantiated first, then your other behaviors
-- need to tell it that they have now been "begun":
--

```

```

-- property myButtonRef, myExpectedMessage, myButtonSprite
--
-- on beginSprite me
--   buttonRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
--   if objectP (buttonRef) then
--     myButtonRef = buttonRef
--   end if
--   -- other stuff
-- end
--
-- If both the above handlers are present, then it doesn't matter which order
-- the sprites appear in the Score.

-- * Calling Objects
-- Non-behavior objects cannot receive events through sendAllSprites. If you
-- want any of your objects to be aware that the button has been activated,
-- your objects should adapt the 'beginSprite' handler above:
--
-- property myButtonRef, myExpectedMessage
--
-- on SetButtonRef me
--   theRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
--   if objectP (theRef) then
--     myButtonRef = theRef
--   end if
--   -- other stuff
-- end
--
-- Note that the object must send the #PushButton_Subscribe call regularly,
-- until it has received a reply.

-- ENABLING AND DISABLING THE BUTTON
-- You may need to disable the button for some reason. For example, the
-- "Jump Back Button" should show a disabled button if no forward navigation has
-- taken place yet, or if the user has returned to the very first frame. The
-- PushButton_ToggleActive handler allows you to set this state with an
-- external call. Use the following syntax:
--
-- sendSprite (<spriteNum>, #PushButton_ToggleActive, trueOrFalse)
--
-- This sets the property myActiveFlag to either TRUE or FALSE. The parameter
-- "trueOrFalse" is optional. If you leave it void, then myActiveFlag simply
-- switches to the opposite boolean value.

-- KILLING OBJECTS
-- You may also wish to include a call and/or handler to allow either the object
-- or the behavior to remove its reference from the other. Your object should
-- call #PushButton_Unsubscribe, and it should handle a #PushButton_ClosingDown
-- call sent by the behavior.
--
-- Examples (for behaviors):
--
-- property myButtonRef
--
-- on endSprite me
--   call (#PushButton_Unsubscribe, myButtonRef, me)
-- end endSprite
--
-- on PushButton_ClosingDown me, buttonRef
--   if buttonRef = myButtonRef then
--     myButtonRef = void
--   end if
-- end

-- KEEPING THE BEHAVIOR PARAMETERS DIALOG SIMPLE
-- This behavior features a GetSuitableMembers handler which returns a list of
-- Cast Members corresponding to a limited number of member types. This means
-- that the Behavior Parameters dialog displays only those members likely to be
-- chosen by the author. Note how it is written to accept a list of types
-- as a parameter. This means that the same handler can serve for different

```

-- member types. I use it once for visual members, and once for sound.

-- The getPropertyDescriptionList handler produces two different
-- Behavior Parameters dialogs, depending on whether any sounds are
-- available.

-- TROUBLESHOOTING

-- If the Rollover Button appears in a MIAW which is not currently the
-- frontWindow, then the user may have to click twice to get the button
-- to function. The first click brings the window to the front, the second
-- triggers the button's handlers.

-- HISTORY --

-- 10 September 1998: Written for the D7 Behaviors Palette by James Newton
-- 2 November 1998: Inter-object communication and myXRayFlag added, all
-- mouse events transferred to the prepareFrame/CheckRollover
-- handler. Notes rewritten.
-- 9 November 1998: enabled/disabled state added
-- 23 February 1999: do/sendAllSprites/call the actorList choice added
-- "on prepareFrame" switched to exitFrame to allow navigation
-- Thanks to Irv Kalb for the inter-object communication handlers
-- 5 January 2000: updated to D8 <km>

-- PROPERTIES --

```
property spriteNum
property mySprite
property myMember
-- author-defined parameters
property myStandardMember
property myRolloverMember
property myMouseDownMember
property myDisabledMember
property myMouseDownSound
property myMouseUpSound
property myActiveFlag
property myXRayFlag
property myMessageType
property myMessage
-- internal properties
property theMouseWasUp
property myMouseDown
property myRollover
-- subscriptions
property mySubscribersList
global gOn
global gStopped
```

-- EVENT HANDLERS --

```
on beginSprite me
  Initialize me
end beginSprite
```

```
on exitFrame me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    if myActiveFlag then CheckForRollover me
  end if
```

```

end exitFrame

on endSprite me
if gOn = TRUE or gStopped = TRUE then
  nothing
else
  -- Inform any subscribed objects that the sprite no longer exists
  call (#PushButton_ClosingDown, mySubscribersList, me, spriteNum, myMessage)
end if

end endSprite

-- CUSTOM HANDLERS --

on Initialize me -- sent by beginSprite
mySprite = sprite(me.spriteNum)
myMember = mySprite.member

-- Error checking: myMessage
repeat while the last char of myMessage = SPACE
  delete the last char of myMessage
end repeat
if not ["do", "no action"].getPos(myMessageType) then
  if myMessage contains SPACE then
    errorAlert(me, #spaceInMessage, myMessage)
  else
    myMessage = symbol (myMessage)
  end if
end if
-- End of error checking

-- Convert properties
myActiveFlag = (myActiveFlag = "Active")
myXRayFlag = (myXRayFlag = "let all mouse events through")

-- Insurance: properties are indeed #members
myStandardMember = value (myStandardMember)
myRolloverMember = value (myRolloverMember)
myMouseDownMember = value (myMouseDownMember)
myDisabledMember = value (myDisabledMember)
myMouseDownSound = value (myMouseDownSound)
myMouseUpSound = value (myMouseUpSound)

if myActiveFlag then
  myMember = myStandardMember
  mySprite.member = myMember
else
  myMember = myDisabledMember
  mySprite.member = myMember
end if

-- Allow other behaviors to subscribe for calls
mySubscribersList = []
sendAllSprites \
(
#PushButton_OpenForBusiness, \
mySubscribersList, \
me, \
spriteNum, \
myMessage \
)
end Initialize

on CheckForRollover me -- sent by prepareFrame
mouseOverMe = rollover (spriteNum)

```

```

if mouseOverMe then
  if not myXRayFlag then
    mouseOverMe = (the mouseMember = myMember)
  end if
end if

if myRollover = mouseOverMe then
  if theMouseWasUp = the mouseUp then
    exit -- Nothing has changed

else -- The mouse has been clicked or released
  theMouseWasUp = the mouseUp
  if mouseOverMe then
    if the mouseUp then
      if myMouseDown then
        Activate me
      else
        -- Mouse was clicked elsewhere then dragged and released over button
        myMember = myRolloverMember
        mySprite.member = myMember
      end if
    else
      -- Button has been clicked on
      ClickOn me
    end if

else -- The mouse is no longer over the button
  if the mouseUp then
    if myMouseDown then
      Disactivate me
    end if
  end if
end if

else -- The rollover state has changed
  set myRollover to mouseOverMe
  if myMouseDown then -- Mouse clicked on this sprite
    if myRollover then
      myMember = myMouseDownMember
      mySprite.member = myMember
    else
      -- Indicate that releasing the mouse button will have no effect
      myMember = myStandardMember
      mySprite.member = myMember
    end if

else -- Sprite has not been clicked
  if not the mouseDown and myRollover then
    myMember = myRolloverMember
    mySprite.member = myMember
  else
    -- No reaction if mouse was clicked elsewhere and dragged to button
    -- or if the mouse is not over the button
    myMember = myStandardMember
    mySprite.member = myMember
  end if
end if
end if
end CheckForRollover

on ClickOn me -- sent by CheckForRollover
  myMouseDown = TRUE
  myMember = myMouseDownMember
  mySprite.member = myMember
  if not voidP(myMouseDownSound) and myMouseDownSound <> #none then
    puppetSound myMouseDownSound
  end if

```

```
updateStage
end ClickOn
```

```
on Activate me -- sent by CheckForRollover
myMouseDown = FALSE
theMouseWasUp = FALSE -- use TRUE if you want the standard state to appear
myMember = myStandardMember
mySprite.member = myMember
if not voidP (myMouseUpSound) and myMouseUpSound <> #none then
  puppetSound myMouseUpSound
end if
updateStage
case myMessageType of
  "do": do myMessage
  "sendAllSprites": sendAllSprites (myMessage, me, spriteNum)
  "call the actorList": call (myMessage, the actorList, me, spriteNum)
end case

call (#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)
end Activate
```

```
on Disactivate me -- sent by CheckForRollover
myMouseDown = FALSE
myMember = myStandardMember
mySprite.member = myMember
end Disactivate
```

-- PUBLIC METHODS (responses to #sendSprite, #sendAllSprites, #call) --

```
on PushButton_ToggleActive me, trueOrFalse
-- called by external Lingo. This handler toggles the property "myActiveFlag"
-- between TRUE and FALSE, or sets it to the value defined by the optional
-- parameter "trueOrFalse".

if voidP (trueOrFalse) then
  myActiveFlag = not myActiveFlag
else
  case ilk (trueOrFalse) of
    #integer: myActiveFlag = trueOrFalse <> 0
    otherwise
      -- Error check
      return #invalidTypeError
  end case
end if

if myActiveFlag then
  myMember = myStandardMember
  mySprite.member = myMember
  call (#PushButton_Enabled, mySubscribersList, me, spriteNum, myMessage)
else
  myMember = myDisabledMember
  mySprite.member = myMember
  call (#PushButton_Disabled, mySubscribersList, me, spriteNum, myMessage)
end if
end PushButton_ToggleActive
```

```
on PushButton_GetReference me, theList
-- Returns a reference to the current behavior. theList is an optional
-- parameter. Use an empty list in a sendAllSprites call to return a
-- list of all "Push Button" behaviors in the current frame. Use
-- an empty linear list to obtain a list of behaviors, or an empty
-- property list to return a list with sprite numbers as the
-- properties and behavior references as the values. Examples :
--
-- put sendAllSprites (#PushButton_GetReference, [])
```

```

-- -- [<offspring "Push Button" 2 2f1b594>]
--
-- put sendAllSprites (#PushButton_GetReference, [:])
-- -- [1: <offspring "Push Button" 2 2f1b594>]
--
-- If you leave "theList" void then the handler will return a reference to
-- the behavior on the given sprite (using sendSprite) or the highest sprite
-- with the behavior (using sendAllSprites). Examples:
--
-- put sendSprite (1, #PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f1b594>
--
-- put sendAllSprites (#PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f0dac0>

```

```

case ilk (theList) of
  #list: theList.append(me)
  #propList: theList.addProp(me.spriteNum, me)
  otherwise
    return me
end case
return theList
end PushButton_GetReference

```

-- INTER-OBJECT COMMUNICATION --

```

on PushButton_Subscribe me, callingBehavior, theMessage
  -- sent by another behavior or an object which needs to receive
  -- PushButton_Activated messages from this behavior. theMessage is an
  -- optional parameter. If not void, it must correspond to the value
  -- for myMessage set in the Behavior Parameters dialog.

```

```

if not voidP(theMessage) then
  if theMessage <> myMessage then
    -- This is not the button the object is looking for
    exit
  end if
end if

```

```

if mySubscribersList.getPos(callingBehavior) then
  -- the object is already on mySubscribersList
  exit
else if objectP(callingBehavior) then
  mySubscribersList.append(callingBehavior)
  return me
end if
end PushButton_Subscribe

```

```

on PushButton_Unsubscribe me, callingBehavior
  mySubscribersList.deleteOne(callingBehavior)
end PushButton_Unsubscribe

```

```

on substituteStrings(me, parentString, childStringList) -----
  -- Sent by errorAlert
  --
  -- * Modifies parentString so that the strings which appear as
  -- properties in childStringList are replaced by the values
  -- associated with those properties.
  --
  -- <childStringList> has the format ["^1": "replacement string"]
  -----

```

```

i = childStringList.count()
repeat while i
  tempString = ""

```

```

dummyString = childStringList.getPropA(i)
replacement = childStringList[i]
lengthAdjust = dummyString.char.count - 1
repeat while TRUE
  position = offset(dummyString, parentString)
  if not position then
    parentString = tempString&parentString
    exit repeat
  else
    if position <> 1 then
      tempString = tempString&parentString.char[1..position - 1]
    end if
    tempString = tempString&replacement
    delete parentString.char[1..position + lengthAdjust]
  end if
end repeat
i = i - 1
end repeat

return parentString
end substituteStrings

```

```
-- ERROR CHECKING --
```

```

on errorAlert me, theError, data
  -- sent by Initialize

```

```

case theError of

  #spaceInMessage:
    if the runmode = "Author" then
      -- Determine the behavior's name
      behaviorName = string (me)
      delete word 1 of behaviorName
      delete the last word of behaviorName
      delete the last word of behaviorName
      tError1 = "BEHAVIOR ERROR: Frame ^0, Sprite ^1"
      tError1 = substituteStrings(me, tError1, ["^0":the frame, "^1":the currentSpriteNum])
      tError2 = "Behavior ^0"
      tError2 = substituteStrings(me, tError2, ["^0":behaviorName])
      tError3 = "This message includes spaces: ^0"
      tError3 = substituteStrings(me, tError3, ["^0":QUOTE & data & QUOTE])
      tError4 = "Only the first word will be used in sendAllSprite calls."
      alert(tError1 & RETURN & RETURN & tError2 & RETURN & RETURN & tError3 & RETURN & RETURN & tError4)
    end if
  end case

end ErrorAlert

```

```
on isOKToAttach (me, aSpriteType, aSpriteNum)
```

```

  tIsOk = 0
  if aSpriteType = #graphic then
    if PermittedMemberTypes().getOne(sprite(aSpriteNum).member.type) > 0 then
      tIsOk = 1
    end if
  end if

  return(tIsOk)
end on

```

```
-- AUTHOR-DEFINED PARAMETERS --
```

on getPropertyDescriptionList me

-- Error check: does current sprite contain appropriate member type?

```
theMember = sprite(the currentSpriteNum).member
memberType = theMember.type
permittedTypes = PermittedMemberTypes(me)
theMemberNum = theMember.number
```

-- Create list of suitable sprite members

```
suitableMembersList = GetSuitableMembers (me, permittedTypes)
```

-- Create modular descriptionList

```
descriptionList = \
[ \
#myStandardMember: \
[ \
#comment: "- GRAPHICS - Standard member for sprite", \
#format: #member, \
#range: suitableMembersList, \
#default: theMember \
], \
#myRolloverMember: \
[ \
#comment: "Rollover member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 1) \
], \
#myMouseDownMember: \
[ \
#comment: "MouseDown member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 2) \
], \
#myDisabledMember: \
[ \
#comment: "Disabled member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 3) \
] \
] \
]
```

-- Check if any sounds are available

```
soundsAvailable = GetSuitableMembers (me, [#sound])
if soundsAvailable.count() then
  soundsAvailable.addAt(1, #none)
  descriptionList.addProp \
(\
#myMouseDownSound, \
[ \
#comment: "- SOUNDS - Sound to play on mouseDown", \
#format: #sound, \
#range: soundsAvailable, \
#default: member (theMemberNum + 4) \
] \
)

  descriptionList.addProp \
(\
#myMouseUpSound, \
[ \
#comment: "Sound to play on mouseUp", \
#format: #sound, \
#range: soundsAvailable, \
#default: member (theMemberNum + 5) \
] \
)
end if
```

```

-- Place remaining properties at the end
descriptionList.addProp \
(\
#myActiveFlag, \
[\
#comment: "- INTERACTION - Button is initially", \
#format: #string, \
#range: ["Active", "Inactive"], \
#default: "Active" \
]\
)
descriptionList.addProp \
(\
#myXRayFlag, \
[\
#comment: "Sprites which cover the button", \
#format: #string, \
#range: ["block all mouse events", "let all mouse events through"], \
#default: FALSE \
]\
)
descriptionList.addProp \
(\
#myMessageType, \
[\
#comment: "Action on mouseUp", \
#format: #string, \
#range: ["do", "sendAllSprites", "call the actorList", "no action"], \
#default: "sendAllSprites" \
]\
)
descriptionList.addProp \
(\
#myMessage, \
[\
#comment: "", \
#format: #string, \
#default: "YourCustomMessage" \
]\
)

return descriptionList
end getPropertyDescriptionList

```

```

on GetSuitableMembers me, permittedTypes
-- Returns a list of all members in all casts
-- corresponding to any of the list of types
cursor 4
suitableMembersList = []
maxCastLib = the number of castLibs
repeat with theCastLib = 1 to maxCastLib
maxMember = the number of members of castLib theCastLib
repeat with memberNumber = 1 to maxMember
theMember = member(memberNumber, theCastLib)
if permittedTypes.getPos(theMember.type) then
if theMember.name = EMPTY then
suitableMembersList.append(theMember)
else
suitableMembersList.append(theMember.name)
end if
end if
end repeat
end repeat
cursor -1
return suitableMembersList
end GetSuitableMembers

```

```

on PermittedMemberTypes me
-- sent by:
-- getBehaviorDescription
-- isOKtoAttach

return [#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, \
#shape, #vectorShape]
end PermittedMemberTypes

```

NumberPad

-- DESCRIPTION --

```

on getBehaviorDescription me
return \
"PUSH BUTTON" & RETURN & RETURN & \
"This behavior sets the member of a sprite depending on the state of the mouse (elsewhere, rollover, mouseDown,
mouseUp)." & RETURN & RETURN & \
"This creates a button which can either initiate actions in other sprites, or provide visual feedback for other behaviors
attached to the same sprite." & RETURN & RETURN & \
"The behavior can be enabled or disabled, using a #PushButton_ToggleActive call to the behavior or the sprite." &
RETURN & RETURN & \
"Two messaging systems are provided:" & RETURN & \
"1) A custom message can sent whenever the Push Button behavior is activated. " & \
"This message can be sent to a Movie Script handler, all sprites, or the actorList. " & \
"The message can also be suppressed." & RETURN & RETURN & \
"2) Objects can 'subscribe' to the behavior in order to receive PushButton_Activated, _Enabled and _Disabled
messages." & \
"A two-way messaging system allows for cleaning up object references before an object is destroyed." & RETURN &
RETURN & \
"The behavior can be set to consider that all sprites in higher channels either block or let through all mouse events. " & \
"If mouse events are allowed to pass, you can place blended sprites above the button to change its color. " & \
"If mouse events are blocked, such translucent sprites provide an alternative method for disabling the button." &
RETURN & RETURN & \
"PERMITTED MEMBER TYPES" & RETURN & \
"[#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, #shape, #vectorShape]" & RETURN & RETURN & \
"PARAMETERS:" & RETURN & \
"* Standard member (when mouse is elsewhere)" & RETURN & \
"* Rollover member" & RETURN & \
"* MouseDown member" & RETURN & \
"* Disabled member" & RETURN & RETURN & \
"Optional parameters:" & RETURN & \
"* MouseDown sound" & RETURN & \
"* MouseUp sound" & RETURN & \
"If members are placed consecutively in the cast in this order then default values can be used to create the button." &
RETURN & RETURN & \
"* Do sprites above the button allow mouse events through?" & RETURN & \
"* Type of message sent on mouseUp: do | sendAllSprites | call the actorList | no action" & RETURN & \
"* Custom Message sent on mouseUp" & RETURN & RETURN & \
"NOTES:" & RETURN & \
"If you use 'do', be sure that you have a handler in a Movie Script that corresponds to the message sent." & RETURN &
RETURN & \
"If you indicate 'no action' then this behavior will simply deal with the different states of the button. " & \
"You can still execute an action on mouseUp by one of two means:" & RETURN & \
"1) Add a behavior with a mouseUp handler to the same sprite (for example, the 'Jump Back Button' behavior" &
RETURN & \
"2) Subscribe an object to the current behavior. " & \
"The object will then receive your Custom Message directly."
end getBehaviorDescription

```

```

on getBehaviorTooltip me
return \
"Use with graphic members." & RETURN & RETURN & \
"Swaps the member of the sprite according to the state of the mouse. " & \
"You can use this dynamic button behavior to play a brief sound on mouseDown and/or mouseUp, send out a custom
message of your choice, and trigger actions of other sprites. " & \

```

"You can also use it to provide visual feedback for other behaviors on the same sprite (for example, Jump to Marker Button). " & \

"This behavior can also interact with custom objects."
end `getBehaviorTooltip`

-- NOTES FOR DEVELOPERS --

-- A BUTTON BEHAVIOR WITH NO MOUSE EVENTS

-- This behavior makes a sprite react when it is clicked on. You might expect
-- to find mouseUp and mouseDown events... but there aren't any. There are no
-- mouseEnter or mouseLeave handlers either, and yet the behavior reacts to
-- rollover.

--

-- The reason for this is that Director is in two minds about intervening
-- sprites. Suppose there is a sprite between the mouse and the button graphic
-- that this behavior is attached to. Such an intervening sprite is opaque
-- for mouseEnter and mouseLeave: the button graphic beneath never receives
-- these events. However, the intervening sprite may be transparent for
-- mouseUp and mouseDown events: if it has no mouseUp/mouseDown handlers
-- attached (through a behavior or Cast Member script), then it lets the button
-- graphic beneath react to these events.

--

-- Relying on mouse events may lead to an ambiguous situation where the button
-- does not indicate that it is active on rollover yet it reacts when it is
-- clicked on.

-- ROLL-YOUR-OWN MOUSE EVENTS

-- To avoid this ambiguity, I let you choose in the Behavior Parameters dialog
-- whether the button can see through intervening sprites (in which case
-- it reacts both to rollover and mouseClicks) or whether it treats intervening
-- sprites as completely opaque (in which case it doesn't react at all).
-- This information is stored in myXRayFlag parameter. In order to provide
-- this choice I had to abandon the mouseEnter and mouseLeave events.

-- Instead, I test the state of where the mouse is and whether it is up or
-- down, on each prepareFrame. If you are using X-ray mode then I test for
-- rollover (spriteNum). If not, I test for (the mouseMember = myMember).
-- This ignores the transparent parts of intervening sprites with matte ink.

-- Both the prepareFrame and the exitFrame handlers are suitable places to do
-- this. I chose to use prepareFrame to run a CheckRollover handler. It is
-- important that actions only take place when the rollover or mouseDown states
-- change. The property myRollover remembers rollover from one frame to the
-- next, and the property myMouseDown remembers the mouseDown state. Together
-- these properties determine whether the member to display should be
-- myRolloverMember, myMouseDownMember or myStandardMember.

-- The theMouseWasUp property has a subtle role. It ensures that the button
-- is not activated if the mouse is clicked elsewhere, then dragged
-- and released over the button. A good part of the CheckForRollover handler
-- deals with this one rare case. The Activate handler sets theMouseWasUp
-- to FALSE. This means that the button will appear in its rollover state
-- after a click. If you prefer to show the standard state after a click,
-- then set theMouseWasUp to TRUE in the Activate handler.

-- TRIGGERING EVENTS

-- A button is not very useful if it can't be used to trigger events
-- elsewhere. I have built in four possibilities in two flavors:
-- 1) sending a custom message to
-- * a Movie Script handler (using a "do" command)
-- * all other sprites (using a "sendAllSprites" command)
-- * objects on the actorList (using a "call ... " & \

"the actorList" command)

-- 2) calling a list of subscribed objects:

-- call (#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)

--

-- Using "do"

-- If you use the "do" command, you must ensure that there is a handler in a

```

-- Movie Script which corresponds to the CustomMessage that you send.
-- If you do not do this, Director will indicate an error
--
-- Using "sendAllSprites"
-- Any sprites that are to be affected when the button is activated must have
-- an 'on YourCustomMessage me, callingBehaviorRef, callingSpriteNum' handler
-- in one of their behaviors.

-- #YourCustomMessage has to be a single word otherwise it can't be
-- converted to a symbol. The getPropertyDescriptionList handler
-- could specify #symbol format. This would clip the author's input at the
-- end of the first word, without warning. I prefer to alert authors that their
-- chosen message is not valid, so I deal with the #symbol conversion myself.
--
-- Calling objects on the actorList
-- The actorList is a convenient place for storing objects without using global
-- variables. You can send a message to all objects on the actorList without
-- having to identify the separate objects. All objects with an appropriate
-- handler will respond to the message. If no objects contain an appropriate
-- handler, the message is simply ignored.

-- The actorList is "local" to each window. Each MIAW has a separate actorList.
-- This means that a button on the stage cannot interact directly with an
-- object in the actorList of a MIAW.
--
-- Calling a list of subscribing behaviors or objects
-- You can also inform your other behaviors that the Push Button behavior has
-- been instantiated. To do this, add the following handler to any behavior
-- that needs to receive PushButton_Activated messages. You can use the
-- parameters 'theSprite' and 'message' to ensure that only the right
-- Push Button is treated.
--
-- property myButtonRef, myExpectedMessage, myButtonSprite
--
-- on PushButton_OpenForBusiness me, theList, buttonRef, theSprite, message
-- if not voidP (message) then
--   if message <> myExpectedMessage then exit -- Sent by the wrong button
-- end if
--   -- Store data concerning the button (this is optional)
--   myButtonRef = buttonRef -- Object reference for the button behavior
--   myButtonSprite = theSprite -- Sprite number of button
--   -- Tell the button behavior to call this object
--   theList.append(me)
--   return theList
-- end PushButton_OpenForBusiness
--
-- If the Push Button behavior is instantiated first, then your other behaviors
-- need to tell it that they have now been "begun":
--
-- property myButtonRef, myExpectedMessage, myButtonSprite
--
-- on beginSprite me
--   buttonRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
--   if objectP (buttonRef) then
--     myButtonRef = buttonRef
--   end if
--   -- other stuff
-- end
--
-- If both the above handlers are present, then it doesn't matter which order
-- the sprites appear in the Score.

-- * Calling Objects
-- Non-behavior objects cannot receive events through sendAllSprites. If you
-- want any of your objects to be aware that the button has been activated,
-- your objects should adapt the 'beginSprite' handler above:
--
-- property myButtonRef, myExpectedMessage
--
-- on SetButtonRef me

```

```

-- theRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
-- if objectP (theRef) then
--   myButtonRef = theRef
-- end if
-- -- other stuff
-- end
--
-- Note that the object must send the #PushButton_Subscribe call regularly,
-- until it has received a reply.

-- ENABLING AND DISABLING THE BUTTON
-- You may need to disable the button for some reason. For example, the
-- "Jump Back Button" should show a disabled button if no forward navigation has
-- taken place yet, or if the user has returned to the very first frame. The
-- PushButton_ToggleActive handler allows you to set this state with an
-- external call. Use the following syntax:
--
-- sendSprite (<spriteNum>, #PushButton_ToggleActive, trueOrFalse)
--
-- This sets the property myActiveFlag to either TRUE or FALSE. The parameter
-- "trueOrFalse" is optional. If you leave it void, then myActiveFlag simply
-- switches to the opposite boolean value.

-- KILLING OBJECTS
-- You may also wish to include a call and/or handler to allow either the object
-- or the behavior to remove its reference from the other. Your object should
-- call #PushButton_Unsubscribe, and it should handle a #PushButton_ClosingDown
-- call sent by the behavior.
--
-- Examples (for behaviors):
--
-- property myButtonRef
--
-- on endSprite me
--   call (#PushButton_Unsubscribe, myButtonRef, me)
-- end endSprite
--
-- on PushButton_ClosingDown me, buttonRef
--   if buttonRef = myButtonRef then
--     myButtonRef = void
--   end if
-- end

-- KEEPING THE BEHAVIOR PARAMETERS DIALOG SIMPLE
-- This behavior features a GetSuitableMembers handler which returns a list of
-- Cast Members corresponding to a limited number of member types. This means
-- that the Behavior Parameters dialog displays only those members likely to be
-- chosen by the author. Note how it is written to accept a list of types
-- as a parameter. This means that the same handler can serve for different
-- member types. I use it once for visual members, and once for sound.

-- The getPropertyDescriptionList handler produces two different
-- Behavior Parameters dialogs, depending on whether any sounds are
-- available.

-- TROUBLESHOOTING
-- If the Rollover Button appears in a MIAW which is not currently the
-- frontWindow, then the user may have to click twice to get the button
-- to function. The first click brings the window to the front, the second
-- triggers the button's handlers.

-- HISTORY --

-- 10 September 1998: Written for the D7 Behaviors Palette by James Newton
-- 2 November 1998: Inter-object communication and myXRayFlag added, all
-- mouse events transferred to the prepareFrame/CheckRollover
-- handler. Notes rewritten.

```

```
-- 9 November 1998: enabled/disabled state added
-- 23 February 1999: do/sendAllSprites/call the actorList choice added
-- "on prepareFrame" switched to exitFrame to allow navigation
-- Thanks to Irv Kalb for the inter-object communication handlers
-- 5 January 2000: updated to D8 <km>
```

```
-- PROPERTIES --
```

```
property spriteNum
property mySprite
property myMember
-- author-defined parameters
property myStandardMember
property myRolloverMember
property myMouseDownMember
property myDisabledMember
property myMouseDownSound
property myMouseUpSound
property myActiveFlag
property myXRayFlag
property myMessageType
property myMessage
-- internal properties
property theMouseWasUp
property myMouseDown
property myRollover
-- subscriptions
property mySubscribersList

global gOn
global gStopped
```

```
-- EVENT HANDLERS --
```

```
on beginSprite me
  Initialize me
end beginSprite
```

```
on exitFrame me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    if myActiveFlag then CheckForRollover me
  end if
end exitFrame
```

```
end exitFrame
```

```
on endSprite me
  if gOn = TRUE or gStopped = TRUE then
    nothing
  else
    -- Inform any subscribed objects that the sprite no longer exists
    call (#PushButton_ClosingDown, mySubscribersList, me, spriteNum, myMessage)
  end if
end endSprite
```

```
end endSprite
```

```
-- CUSTOM HANDLERS --
```

```
on Initialize me -- sent by beginSprite
  mySprite = sprite(me.spriteNum)
  myMember = mySprite.member
```

```

-- Error checking: myMessage
repeat while the lastchar of myMessage = SPACE
  delete the last char of myMessage
end repeat
if not ["do", "no action"].getPos(myMessageType) then
  if myMessage contains SPACE then
    errorAlert(me, #spaceInMessage, myMessage)
  else
    myMessage = symbol (myMessage)
  end if
end if
-- End of error checking

-- Convert properties
myActiveFlag = (myActiveFlag = "Active")
myXRayFlag = (myXRayFlag = "let all mouse events through")

-- Insurance: properties are indeed #members
myStandardMember = value (myStandardMember)
myRolloverMember = value (myRolloverMember)
myMouseDownMember = value (myMouseDownMember)
myDisabledMember = value (myDisabledMember)
myMouseDownSound = value (myMouseDownSound)
myMouseUpSound = value (myMouseUpSound)

if myActiveFlag then
  myMember = myStandardMember
  mySprite.member = myMember
else
  myMember = myDisabledMember
  mySprite.member = myMember
end if

-- Allow other behaviors to subscribe for calls
mySubscribersList = []
sendAllSprites \
(
#PushButton_OpenForBusiness, \
mySubscribersList, \
me, \
spriteNum, \
myMessage \
)
end Initialize

on CheckForRollover me -- sent by prepareFrame
mouseOverMe = rollover (spriteNum)
if mouseOverMe then
  if not myXRayFlag then
    mouseOverMe = (the mouseMember = myMember)
  end if
end if

if myRollover = mouseOverMe then
  if theMouseWasUp = the mouseUp then
    exit -- Nothing has changed
  else -- The mouse has been clicked or released
    theMouseWasUp = the mouseUp
    if mouseOverMe then
      if the mouseUp then
        if myMouseDown then
          Activate me
        else
          -- Mouse was clicked elsewhere then dragged and released over button
          myMember = myRolloverMember
          mySprite.member = myMember
        end if
      end if
    end if
  end if
end if

```

```

else
  -- Button has been clicked on
  ClickOn me
end if

else -- The mouse is no longer over the button
  if the mouseUp then
    if myMouseDown then
      Disactivate me
    end if
  end if
end if
end if

else -- The rollover state has changed
set myRollover to mouseOverMe
if myMouseDown then -- Mouse clicked on this sprite
  if myRollover then
    myMember = myMouseDownMember
    mySprite.member = myMember
  else
    -- Indicate that releasing the mouse button will have no effect
    myMember = myStandardMember
    mySprite.member = myMember
  end if
else -- Sprite has not been clicked
  if not the mouseDown and myRollover then
    myMember = myRolloverMember
    mySprite.member = myMember
  else
    -- No reaction if mouse was clicked elsewhere and dragged to button
    -- or if the mouse is not over the button
    myMember = myStandardMember
    mySprite.member = myMember
  end if
end if
end if
end CheckForRollover

on ClickOn me -- sent by CheckForRollover
  myMouseDown = TRUE
  myMember = myMouseDownMember
  mySprite.member = myMember
  if not voidP(myMouseDownSound) and myMouseDownSound <> #none then
    puppetSound myMouseDownSound
  end if
  updateStage
end ClickOn

on Activate me -- sent by CheckForRollover
  myMouseDown = FALSE
  theMouseWasUp = FALSE -- use TRUE if you want the standard state to appear
  myMember = myStandardMember
  mySprite.member = myMember
  if not voidP(myMouseUpSound) and myMouseUpSound <> #none then
    puppetSound myMouseUpSound
  end if
  updateStage
  case myMessageType of
    "do": do myMessage
    "sendAllSprites": sendAllSprites(myMessage, me, spriteNum)
    "call the actorList": call(myMessage, the actorList, me, spriteNum)
  end case

  call(#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)
end Activate

```

```

on Disactivate me -- sent by CheckForRollover
myMouseDown = FALSE
myMember = myStandardMember
mySprite.member = myMember
end Disactivate

-- PUBLIC METHODS (responses to #sendSprite, #sendAllSprites, #call) --

on PushButton_ToggleActive me, trueOrFalse
-- called by external Lingo. This handler toggles the property "myActiveFlag"
-- between TRUE and FALSE, or sets it to the value defined by the optional
-- parameter "trueOrFalse".

if voidP(trueOrFalse) then
myActiveFlag = not myActiveFlag
else
case ilk (trueOrFalse) of
#integer: myActiveFlag = trueOrFalse <> 0
otherwise
-- Error check
return #invalidTypeError
end case
end if

if myActiveFlag then
myMember = myStandardMember
mySprite.member = myMember
call (#PushButton_Enabled, mySubscribersList, me, spriteNum, myMessage)
else
myMember = myDisabledMember
mySprite.member = myMember
call (#PushButton_Disabled, mySubscribersList, me, spriteNum, myMessage)
end if
end PushButton_ToggleActive

on PushButton_GetReference me, theList
-- Returns a reference to the current behavior. theList is an optional
-- parameter. Use an empty list in a sendAllSprites call to return a
-- list of all "Push Button" behaviors in the current frame. Use
-- an empty linear list to obtain a list of behaviors, or an empty
-- property list to return a list with sprite numbers as the
-- properties and behavior references as the values. Examples :
--
-- put sendAllSprites (#PushButton_GetReference, [])
-- -- [<offspring "Push Button" 2 2f1b594>]
--
-- put sendAllSprites (#PushButton_GetReference, [:])
-- -- [1: <offspring "Push Button" 2 2f1b594>]
--
-- If you leave "theList" void then the handler will return a reference to
-- the behavior on the given sprite (using sendSprite) or the highest sprite
-- with the behavior (using sendAllSprites). Examples:
--
-- put sendSprite (1, #PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f1b594>
--
-- put sendAllSprites (#PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f0dac0>

case ilk (theList) of
#list: theList.append(me)
#propList: theList.addProp(me.spriteNum, me)
otherwise
return me
end case

```

```
return theList
end PushButton_GetReference
```

```
-- INTER-OBJECT COMMUNICATION --
```

```
on PushButton_Subscribe me, callingBehavior, theMessage
-- sent by another behavior or an object which needs to receive
-- PushButton_Activated messages from this behavior. theMessage is an
-- optional parameter. If not void, it must correspond to the value
-- for myMessage set in the Behavior Parameters dialog.
```

```
if not voidP(theMessage) then
if theMessage <> myMessage then
-- This is not the button the object is looking for
exit
end if
end if
```

```
if mySubscribersList.getPos(callingBehavior) then
-- the object is already on mySubscribersList
exit
else if objectP(callingBehavior) then
mySubscribersList.append(callingBehavior)
return me
end if
end PushButton_Subscribe
```

```
on PushButton_Unsubscribe me, callingBehavior
mySubscribersList.deleteOne(callingBehavior)
end PushButton_Unsubscribe
```

```
on substituteStrings(me, parentString, childStringList) -----
-- Sent by errorAlert
--
-- * Modifies parentString so that the strings which appear as
-- properties in childStringList are replaced by the values
-- associated with those properties.
--
-- <childStringList> has the format ["^1": "replacement string"]
-----
```

```
i = childStringList.count()
repeat while i
tempString = ""
dummyString = childStringList.getPropA(i)
replacement = childStringList[i]
lengthAdjust = dummyString.char.count - 1
repeat while TRUE
position = offset(dummyString, parentString)
if not position then
parentString = tempString&parentString
exit repeat
else
if position <> 1 then
tempString = tempString&parentString.char[1..position - 1]
end if
tempString = tempString&replacement
delete parentString.char[1..position + lengthAdjust]
end if
end repeat
i = i - 1
end repeat

return parentString
end substituteStrings
```

```
-- ERROR CHECKING --
```

```
on errorAlert me, theError, data  
-- sent by Initialize
```

```
case theError of
```

```
  #spaceInMessage:  
  if the runmode = "Author" then  
    -- Determine the behavior's name  
    behaviorName = string (me)  
    delete word 1 of behaviorName  
    delete the last word of behaviorName  
    delete the last word of behaviorName  
    tError1 = "BEHAVIOR ERROR: Frame ^0, Sprite ^1"  
    tError1 = substituteStrings(me, tError1, ["^0":the frame, "^1":the currentSpriteNum])  
    tError2 = "Behavior ^0"  
    tError2 = substituteStrings(me, tError2, ["^0":behaviorName])  
    tError3 = "This message includes spaces: ^0"  
    tError3 = substituteStrings(me, tError3, ["^0":QUOTE & data & QUOTE])  
    tError4 = "Only the first word will be used in sendAllSprite calls."  
    alert(tError1 & RETURN & RETURN & tError2 & RETURN & RETURN & tError3 & RETURN & RETURN & tError4)  
  end if  
end case
```

```
end ErrorAlert
```

```
on isOKToAttach (me, aSpriteType, aSpriteNum)
```

```
  tIsOk = 0  
  if aSpriteType = #graphic then  
    if PermittedMemberTypes().getOne(sprite(aSpriteNum).member.type) > 0 then  
      tIsOk = 1  
    end if  
  end if  
  
  return(tIsOk)  
end on
```

```
-- AUTHOR-DEFINED PARAMETERS --
```

```
on getPropertyDescriptionList me
```

```
  -- Error check: does current sprite contain appropriate member type?  
  theMember = sprite(the currentSpriteNum).member  
  memberType = theMember.type  
  permittedTypes = PermittedMemberTypes(me)  
  theMemberNum = theMember.number  
  
  -- Create list of suitable sprite members  
  suitableMembersList = GetSuitableMembers (me, permittedTypes)  
  
  -- Create modular descriptionList  
  descriptionList = \  
  [ \  
  #myStandardMember: \  
  [ \  
  #comment: "- GRAPHICS - Standard member for sprite", \  
  #format: #member, \  
  #range: suitableMembersList, \  
  #default: theMember \  
  ], \  
  ]
```

```

#myRolloverMember: \
[ \
#comment: "Rollover member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 1) \
], \
#myMouseDownMember: \
[ \
#comment: "MouseDown member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 2) \
], \
#myDisabledMember: \
[ \
#comment: "Disabled member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 3) \
] \
] \
]

-- Check if any sounds are available
soundsAvailable = GetSuitableMembers (me, [#sound])
if soundsAvailable.count() then
    soundsAvailable.addAt(1, #none)
    descriptionList.addProp \
( \
#myMouseDownSound, \
[ \
#comment: "- SOUNDS - Sound to play on mouseDown", \
#format: #sound, \
#range: soundsAvailable, \
#default: member (theMemberNum + 4) \
] \
)

    descriptionList.addProp \
( \
#myMouseUpSound, \
[ \
#comment: "Sound to play on mouseUp", \
#format: #sound, \
#range: soundsAvailable, \
#default: member (theMemberNum + 5) \
] \
)
end if

-- Place remaining properties at the end
descriptionList.addProp \
( \
#myActiveFlag, \
[ \
#comment: "- INTERACTION - Button is initially", \
#format: #string, \
#range: ["Active", "Inactive"], \
#default: "Active" \
] \
)
descriptionList.addProp \
( \
#myXRayFlag, \
[ \
#comment: "Sprites which cover the button", \
#format: #string, \
#range: ["block all mouse events", "let all mouse events through"], \
#default: FALSE \
] \
)

```

```

)
descriptionList.addProp \
(
#myMessageType, \
[ \
#comment: "Action on mouseUp", \
#format: #string, \
#range: ["do", "sendAllSprites", "call the actorList", "no action"], \
#default: "sendAllSprites" \
] \
)
descriptionList.addProp \
(
#myMessage, \
[ \
#comment: "", \
#format: #string, \
#default: "YourCustomMessage" \
] \
)
return descriptionList
end getPropertyDescriptionList

```

```

on GetSuitableMembers me, permittedTypes
-- Returns a list of all members in all casts
-- corresponding to any of the list of types
cursor 4
suitableMembersList = []
maxCastLib = the number of castLibs
repeat with theCastLib = 1 to maxCastLib
maxMember = the number of members of castLib theCastLib
repeat with memberNumber = 1 to maxMember
theMember = member(memberNumber, theCastLib)
if permittedTypes.getPos(theMember.type) then
if theMember.name = EMPTY then
suitableMembersList.append(theMember)
else
suitableMembersList.append(theMember.name)
end if
end if
end repeat
end repeat
cursor -1
return suitableMembersList
end GetSuitableMembers

```

```

on PermittedMemberTypes me
-- sent by:
-- getBehaviorDescription
-- isOKtoAttach

return [#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, \
#shape, #vectorShape]
end PermittedMemberTypes

```

OnOff

```
-- DESCRIPTION --
```

```

on getBehaviorDescription me
return \
"PUSH BUTTON" & RETURN & RETURN & \
"This behavior sets the member of a sprite depending on the state of the mouse (elsewhere, rollover, mouseDown, mouseUp)." & RETURN & RETURN & \

```

```

"This creates a button which can either initiate actions in other sprites, or provide visual feedback for other behaviors
attached to the same sprite." & RETURN & RETURN & \
"The behavior can be enabled or disabled, using a #PushButton_ToggleActive call to the behavior or the sprite." &
RETURN & RETURN & \
"Two messaging systems are provided:" & RETURN & \
"1) A custom message can sent whenever the Push Button behavior is activated. " & \
"This message can be sent to a Movie Script handler, all sprites, or the actorList. " & \
"The message can also be suppressed." & RETURN & RETURN & \
"2) Objects can 'subscribe' to the behavior in order to receive PushButton_Activated, _Enabled and _Disabled
messages. " & \
"A two-way messaging system allows for cleaning up object references before an object is destroyed." & RETURN &
RETURN & \
"The behavior can be set to consider that all sprites in higher channels either block or let through all mouse events. " & \
"If mouse events are allowed to pass, you can place blended sprites above the button to change its color. " & \
"If mouse events are blocked, such translucent sprites provide an alternative method for disabling the button." &
RETURN & RETURN & \
"PERMITTED MEMBER TYPES" & RETURN & \
"[#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, #shape, #vectorShape]" & RETURN & RETURN & \
"PARAMETERS:" & RETURN & \
"* Standard member (when mouse is elsewhere)" & RETURN & \
"* Rollover member" & RETURN & \
"* MouseDown member" & RETURN & \
"* Disabled member" & RETURN & RETURN & \
"Optional parameters:" & RETURN & \
"* MouseDown sound" & RETURN & \
"* MouseUp sound" & RETURN & \
"If members are placed consecutively in the cast in this order then default values can be used to create the button." &
RETURN & RETURN & \
"* Do sprites above the button allow mouse events through?" & RETURN & \
"* Type of message sent on mouseUp: do | sendAllSprites | call the actorList | no action" & RETURN & \
"* Custom Message sent on mouseUp" & RETURN & RETURN & \
"NOTES:" & RETURN & \
"If you use 'do', be sure that you have a handler in a Movie Script that corresponds to the message sent." & RETURN &
RETURN & \
"If you indicate 'no action' then this behavior will simply deal with the different states of the button. " & \
"You can still execute an action on mouseUp by one of two means:" & RETURN & \
"1) Add a behavior with a mouseUp handler to the same sprite (for example, the 'Jump Back Button' behavior" &
RETURN & \
"2) Subscribe an object to the current behavior. " & \
" The object will then receive your Custom Message directly."
end getBehaviorDescription

```

```

on getBehaviorTooltip me
return \
"Use with graphic members." & RETURN & RETURN & \
"Swaps the member of the sprite according to the state of the mouse. " & \
"You can use this dynamic button behavior to play a brief sound on mouseDown and/or mouseUp, send out a custom
message of your choice, and trigger actions of other sprites. " & \
"You can also use it to provide visual feedback for other behaviors on the same sprite (for example, Jump to Marker
Button). " & \
"This behavior can also interact with custom objects."
end getBehaviorTooltip

```

-- NOTES FOR DEVELOPERS --

```

-- A BUTTON BEHAVIOR WITH NO MOUSE EVENTS
-- This behavior makes a sprite react when it is clicked on. You might expect
-- to find mouseUp and mouseDown events... but there aren't any. There are no
-- mouseEnter or mouseLeave handlers either, and yet the behavior reacts to
-- rollover.
--
-- The reason for this is that Director is in two minds about intervening
-- sprites. Suppose there is a sprite between the mouse and the button graphic
-- that this behavior is attached to. Such an intervening sprite is opaque
-- for mouseEnter and mouseLeave: the button graphic beneath never receives
-- these events. However, the intervening sprite may be transparent for
-- mouseUp and mouseDown events: if it has no mouseUp/mouseDown handlers

```

```
-- attached (through a behavior or Cast Member script), then it lets the button
-- graphic beneath react to these events.
--
-- Relying on mouse events may lead to an ambiguous situation where the button
-- does not indicate that it is active on rollover yet it reacts when it is
-- clicked on.

-- ROLL-YOUR-OWN MOUSE EVENTS
-- To avoid this ambiguity, I let you choose in the Behavior Parameters dialog
-- whether the button can see through intervening sprites (in which case
-- it reacts both to rollover and mouseClicks) or whether it treats intervening
-- sprites as completely opaque (in which case it doesn't react at all).
-- This information is stored in myXRayFlag parameter. In order to provide
-- this choice I had to abandon the mouseEnter and mouseLeave events.

-- Instead, I test the state of where the mouse is and whether it is up or
-- down, on each prepareFrame. If you are using X-ray mode then I test for
-- rollover (spriteNum). If not, I test for (the mouseMember = myMember).
-- This ignores the transparent parts of intervening sprites with matte ink.

-- Both the prepareFrame and the exitFrame handlers are suitable places to do
-- this. I chose to use prepareFrame to run a CheckRollover handler. It is
-- important that actions only take place when the rollover or mouseDown states
-- change. The property myRollover remembers rollover from one frame to the
-- next, and the property myMouseDown remembers the mouseDown state. Together
-- these properties determine whether the member to display should be
-- myRolloverMember, myMouseDownMember or myStandardMember.

-- The theMouseWasUp property has a subtle role. It ensures that the button
-- is not activated if the mouse is clicked elsewhere, then dragged
-- and released over the button. A good part of the CheckForRollover handler
-- deals with this one rare case. The Activate handler sets theMouseWasUp
-- to FALSE. This means that the button will appear in its rollover state
-- after a click. If you prefer to show the standard state after a click,
-- then set theMouseWasUp to TRUE in the Activate handler.

-- TRIGGERING EVENTS
-- A button is not very useful if it can't be used to trigger events
-- elsewhere. I have built in four possibilities in two flavors:
-- 1) sending a custom message to
--   * a Movie Script handler (using a "do" command)
--   * all other sprites (using a "sendAllSprites" command)
--   * objects on the actorList (using a "call ... " & \
--     "the actorList" command)
-- 2) calling a list of subscribed objects:
--   call (#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)
--
-- Using "do"
-- If you use the "do" command, you must ensure that there is a handler in a
-- Movie Script which corresponds to the CustomMessage that you send.
-- If you do not do this, Director will indicate an error
--
-- Using "sendAllSprites"
-- Any sprites that are to be affected when the button is activated must have
-- an 'on YourCustomMessage me, callingBehaviorRef, callingSpriteNum' handler
-- in one of their behaviors.

-- #YourCustomMessage has to be a single word otherwise it can't be
-- converted to a symbol. The getPropertyDescriptionList handler
-- could specify #symbol format. This would clip the author's input at the
-- end of the first word, without warning. I prefer to alert authors that their
-- chosen message is not valid, so I deal with the #symbol conversion myself.
--
-- Calling objects on the actorList
-- The actorList is a convenient place for storing objects without using global
-- variables. You can send a message to all objects on the actorList without
-- having to identify the separate objects. All objects with an appropriate
-- handler will respond to the message. If no objects contain an appropriate
-- handler, the message is simply ignored.
```

```

-- The actorList is "local" to each window. Each MIAW has a separate actorList.
-- This means that a button on the stage cannot interact directly with an
-- object in the actorList of a MIAW.
--
-- Calling a list of subscribing behaviors or objects
-- You can also inform your other behaviors that the Push Button behavior has
-- been instantiated. To do this, add the following handler to any behavior
-- that needs to receive PushButton_Activated messages. You can use the
-- parameters 'theSprite' and 'message' to ensure that only the right
-- Push Button is treated.
--
-- property myButtonRef, myExpectedMessage, myButtonSprite
--
-- on PushButton_OpenForBusiness me, theList, buttonRef, theSprite, message
--   if not voidP (message) then
--     if message <> myExpectedMessage then exit -- Sent by the wrong button
--   end if
--   -- Store data concerning the button (this is optional)
--   myButtonRef = buttonRef -- Object reference for the button behavior
--   myButtonSprite = theSprite -- Sprite number of button
--   -- Tell the button behavior to call this object
--   theList.append(me)
--   return theList
-- end PushButton_OpenForBusiness
--
-- If the Push Button behavior is instantiated first, then your other behaviors
-- need to tell it that they have now been "begun":
--
-- property myButtonRef, myExpectedMessage, myButtonSprite
--
-- on beginSprite me
--   buttonRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
--   if objectP (buttonRef) then
--     myButtonRef = buttonRef
--   end if
--   -- other stuff
-- end
--
-- If both the above handlers are present, then it doesn't matter which order
-- the sprites appear in the Score.

-- * Calling Objects
-- Non-behavior objects cannot receive events through sendAllSprites. If you
-- want any of your objects to be aware that the button has been activated,
-- your objects should adapt the 'beginSprite' handler above:
--
-- property myButtonRef, myExpectedMessage
--
-- on SetButtonRef me
--   theRef = sendAllSprites (#PushButton_Subscribe, me, myExpectedMessage)
--   if objectP (theRef) then
--     myButtonRef = theRef
--   end if
--   -- other stuff
-- end
--
-- Note that the object must send the #PushButton_Subscribe call regularly,
-- until it has received a reply.

-- ENABLING AND DISABLING THE BUTTON
-- You may need to disable the button for some reason. For example, the
-- "Jump Back Button" should show a disabled button if no forward navigation has
-- taken place yet, or if the user has returned to the very first frame. The
-- PushButton_ToggleActive handler allows you to set this state with an
-- external call. Use the following syntax:
--
-- sendSprite (<spriteNum>, #PushButton_ToggleActive, trueOrFalse)
--
-- This sets the property myActiveFlag to either TRUE or FALSE. The parameter
-- "trueOrFalse" is optional. If you leave it void, then myActiveFlag simply

```

```

-- switches to the opposite boolean value.

-- KILLING OBJECTS
-- You may also wish to include a call and/or handler to allow either the object
-- or the behavior to remove its reference from the other. Your object should
-- call #PushButton_Unsubscribe, and it should handle a #PushButton_ClosingDown
-- call sent by the behavior.
--
-- Examples (for behaviors):
--
--   property myButtonRef
--
--   on endSprite me
--     call (#PushButton_Unsubscribe, my ButtonRef, me)
--   end endSprite
--
--   on PushButton_ClosingDown me, buttonRef
--     if buttonRef = myButtonRef then
--       myButtonRef = void
--     end if
--   end

-- KEEPING THE BEHAVIOR PARAMETERS DIALOG SIMPLE
-- This behavior features a GetSuitableMembers handler which returns a list of
-- Cast Members corresponding to a limited number of member types. This means
-- that the Behavior Parameters dialog displays only those members likely to be
-- chosen by the author. Note how it is written to accept a list of types
-- as a parameter. This means that the same handler can serve for different
-- member types. I use it once for visual members, and once for sound.

-- The getPropertyDescriptionList handler produces two different
-- Behavior Parameters dialogs, depending on whether any sounds are
-- available.

-- TROUBLESHOOTING
-- If the Rollover Button appears in a MIAW which is not currently the
-- frontWindow, then the user may have to click twice to get the button
-- to function. The first click brings the window to the front, the second
-- triggers the button's handlers.

-- HISTORY --

-- 10 September 1998: Written for the D7 Behaviors Palette by James Newton
-- 2 November 1998: Inter-object communication and myXRayFlag added, all
-- mouse events tranferred to the prepareFrame/CheckRollover
-- handler. Notes rewritten.
-- 9 November 1998: enabled/disabled state added
-- 23 February 1999: do/sendAllSprites/call the actorList choice added
-- "on prepareFrame" switched to exitFrame to allow navigation
-- Thanks to Irv Kalb for the inter-object communication handlers
-- 5 January 2000: updated to D8 <km>

-- PROPERTIES --

property spriteNum
property mySprite
property myMember
-- author-defined parameters
property myStandardMember
property myRolloverMember
property myMouseDownMember
property myDisabledMember
property myMouseDownSound
property myMouseUpSound
property myActiveFlag

```

```

property myXRayFlag
property myMessageType
property myMessage
-- internal properties
property theMouseWasUp
property myMouseDown
property myRollover
-- subscriptions
property mySubscribersList

-- EVENT HANDLERS --

on beginSprite me
  Initialize me
end beginSprite

on exitFrame me
  if myActiveFlag then CheckForRollover me
end exitFrame

on endSprite me
  -- Inform any subscribed objects that the sprite no longer exists
  call (#PushButton_ClosingDown, mySubscribersList, me, spriteNum, myMessage)
end endSprite

-- CUSTOM HANDLERS --

on Initialize me -- sent by beginSprite
  mySprite = sprite(me.spriteNum)
  myMember = mySprite.member

  -- Error checking: myMessage
  repeat while the last char of myMessage = SPACE
    delete the last char of myMessage
  end repeat
  if not ["do", "no action"].getPos(myMessageType) then
    if myMessage contains SPACE then
      errorAlert(me, #spaceInMessage, myMessage)
    else
      myMessage = symbol(myMessage)
    end if
  end if
  -- End of error checking

  -- Convert properties
  myActiveFlag = (myActiveFlag = "Active")
  myXRayFlag = (myXRayFlag = "let all mouse events through")

  -- Insurance: properties are indeed #members
  myStandardMember = value(myStandardMember)
  myRolloverMember = value(myRolloverMember)
  myMouseDownMember = value(myMouseDownMember)
  myDisabledMember = value(myDisabledMember)
  myMouseDownSound = value(myMouseDownSound)
  myMouseUpSound = value(myMouseUpSound)

  if myActiveFlag then
    myMember = myStandardMember
    mySprite.member = myMember
  else
    myMember = myDisabledMember
    mySprite.member = myMember
  end if

```

```

-- Allow other behaviors to subscribe for calls
mySubscribersList = []
sendAllSprites \
(
#PushButton_OpenForBusiness, \
mySubscribersList, \
me, \
spriteNum, \
myMessage \
)
end Initialize

on CheckForRollover me -- sent by prepareFrame
mouseOverMe = rollover (spriteNum)
if mouseOverMe then
  if not myXRayFlag then
    mouseOverMe = (the mouseMember = myMember)
  end if
end if

if myRollover = mouseOverMe then
  if theMouseWasUp = the mouseUp then
    exit -- Nothing has changed

  else -- The mouse has been clicked or released
    theMouseWasUp = the mouseUp
    if mouseOverMe then
      if the mouseUp then
        if myMouseDown then
          Activate me
        else
          -- Mouse was clicked elsewhere then dragged and released over button
          myMember = myRolloverMember
          mySprite.member = myMember
        end if
      else
        -- Button has been clicked on
        ClickOn me
      end if
    else -- The mouse is no longer over the button
      if the mouseUp then
        if myMouseDown then
          Disactivate me
        end if
      end if
    end if
  end if

  else -- The rollover state has changed
    set myRollover to mouseOverMe
    if myMouseDown then -- Mouse clicked on this sprite
      if myRollover then
        myMember = myMouseDownMember
        mySprite.member = myMember
      else
        -- Indicate that releasing the mouse button will have no effect
        myMember = myStandardMember
        mySprite.member = myMember
      end if
    else -- Sprite has not been clicked
      if not the mouseDown and myRollover then
        myMember = myRolloverMember
        mySprite.member = myMember
      else
        -- No reaction if mouse was clicked elsewhere and dragged to button
        -- or if the mouse is not over the button

```

```

    myMember = myStandardMember
    mySprite.member = myMember
end if
end if
end if
end CheckForRollover

```

```

on ClickOn me -- sent by CheckForRollover
    myMouseDown = TRUE
    myMember = myMouseDownMember
    mySprite.member = myMember
    if not voidP (myMouseDownSound) and myMouseDownSound <> #none then
        puppetSound myMouseDownSound
    end if
    updateStage
end ClickOn

```

```

on Activate me -- sent by CheckForRollover
    myMouseDown = FALSE
    theMouseWasUp = FALSE -- use TRUE if you want the standard state to appear
    myMember = myStandardMember
    mySprite.member = myMember
    if not voidP (myMouseUpSound) and myMouseUpSound <> #none then
        puppetSound myMouseUpSound
    end if
    updateStage
    case myMessageType of
        "do": do myMessage
        "sendAllSprites": sendAllSprites (myMessage, me, spriteNum)
        "call the actorList": call (myMessage, the actorList, me, spriteNum)
    end case

    call (#PushButton_Activated, mySubscribersList, me, spriteNum, myMessage)
end Activate

```

```

on Disactivate me -- sent by CheckForRollover
    myMouseDown = FALSE
    myMember = myStandardMember
    mySprite.member = myMember
end Disactivate

```

-- PUBLIC METHODS (responses to #sendSprite, #sendAllSprites, #call) --

```

on PushButton_ToggleActive me, trueOrFalse
    -- called by external Lingo. This handler toggles the property "myActiveFlag"
    -- between TRUE and FALSE, or sets it to the value defined by the optional
    -- parameter "trueOrFalse".

    if voidP (trueOrFalse) then
        myActiveFlag = not myActiveFlag
    else
        case ilk (trueOrFalse) of
            #integer: myActiveFlag = trueOrFalse <> 0
            otherwise
                -- Error check
                return #invalidTypeError
        end case
    end if

    if myActiveFlag then
        myMember = myStandardMember
        mySprite.member = myMember
        call (#PushButton_Enabled, mySubscribersList, me, spriteNum, myMessage)
    else

```

```

myMember      = myDisabledMember
mySprite.member = myMember
call (#PushButton_Disabled, mySubscribersList, me, spriteNum, myMessage)
end if
end PushButton_ToggleActive

```

```

on PushButton_GetReference me, theList
-- Returns a reference to the current behavior. theList is an optional
-- parameter. Use an empty list in a sendAllSprites call to return a
-- list of all "Push Button" behaviors in the current frame. Use
-- an empty linear list to obtain a list of behaviors, or an empty
-- property list to return a list with sprite numbers as the
-- properties and behavior references as the values. Examples :
--
-- put sendAllSprites (#PushButton_GetReference, [])
-- -- [<offspring "Push Button" 2 2f1b594>]
--
-- put sendAllSprites (#PushButton_GetReference, [:])
-- -- [1: <offspring "Push Button" 2 2f1b594>]
--
-- If you leave "theList" void then the handler will return a reference to
-- the behavior on the given sprite (using sendSprite) or the highest sprite
-- with the behavior (using sendAllSprites). Examples:
--
-- put sendSprite (1, #PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f1b594>
--
-- put sendAllSprites (#PushButton_GetReference)
-- -- <offspring "Push Button" 2 2f0dac0>

```

```

case ilk (theList) of
#list: theList.append(me)
#propList: theList.addProp(me.spriteNum, me)
otherwise
return me
end case
return theList
end PushButton_GetReference

```

-- INTER-OBJECT COMMUNICATION --

```

on PushButton_Subscribe me, callingBehavior, theMessage
-- sent by another behavior or an object which needs to receive
-- PushButton_Activated messages from this behavior. theMessage is an
-- optional parameter. If not void, it must correspond to the value
-- for myMessage set in the Behavior Parameters dialog.

```

```

if not voidP (theMessage) then
if theMessage <> myMessage then
-- This is not the button the object is looking for
exit
end if
end if

```

```

if mySubscribersList.getPos(callingBehavior) then
-- the object is already on mySubscribersList
exit
else if objectP (callingBehavior) then
mySubscribersList.append (callingBehavior)
return me
end if
end PushButton_Subscribe

```

```

on PushButton_Unsubscribe me, callingBehavior
mySubscribersList.deleteOne(callingBehavior)
end PushButton_Unsubscribe

```

```

on substituteStrings(me, parentString, childStringList) -----
-- Sent by errorAlert
--
-- * Modifies parentString so that the strings which appear as
-- properties in childStringList are replaced by the values
-- associated with those properties.
--
-- <childStringList> has the format ["^1": "replacement string"]
-----

i = childStringList.count()
repeat while i
tempString = ""
dummyString = childStringList.getPropA(i)
replacement = childStringList[i]
lengthAdjust = dummyString.char.count - 1
repeat while TRUE
position = offset(dummyString, parentString)
if not position then
parentString = tempString&parentString
exit repeat
else
if position <> 1 then
tempString = tempString&parentString.char[1..position - 1]
end if
tempString = tempString&replacement
delete parentString.char[1..position + lengthAdjust]
end if
end repeat
i = i - 1
end repeat

return parentString
end substituteStrings

-- ERROR CHECKING --

on errorAlert me, theError, data
-- sent by Initialize

case theError of

#spaceInMessage:
if the runmode = "Author" then
-- Determine the behavior's name
behaviorName = string (me)
delete word 1 of behaviorName
delete the last word of behaviorName
delete the last word of behaviorName
tError1 = "BEHAVIOR ERROR: Frame ^0, Sprite ^1"
tError1 = substituteStrings(me, tError1, ["^0":the frame, "^1":the currentSpriteNum])
tError2 = "Behavior ^0"
tError2 = substituteStrings(me, tError2, ["^0":behaviorName])
tError3 = "This message includes spaces: ^0"
tError3 = substituteStrings(me, tError3, ["^0":QUOTE & data & QUOTE])
tError4 = "Only the first word will be used in sendAllSprite calls."
alert(tError1 & RETURN & RETURN & tError2 & RETURN & RETURN & tError3 & RETURN & RETURN & tError4)
end if
end case

end ErrorAlert

```

```

on isOKToAttach (me, aSpriteType, aSpriteNum)

  tIsOk = 0
  if aSpriteType = #graphic then
    if PermittedMemberTypes().getOne(sprite(aSpriteNum).member.type) > 0 then
      tIsOk = 1
    end if
  end if

  return(tIsOk)
end on

```

-- AUTHOR-DEFINED PARAMETERS --

```

on getPropertyDescriptionList me

  -- Error check: does current sprite contain appropriate member type?
  theMember = sprite(the currentSpriteNum).member
  memberType = theMember.type
  permittedTypes = PermittedMemberTypes(me)
  theMemberNum = theMember.number

  -- Create list of suitable sprite members
  suitableMembersList = GetSuitableMembers (me, permittedTypes)

  -- Create modular descriptionList
  descriptionList = \
[ \
#myStandardMember: \
[ \
#comment: "- GRAPHICS - Standard member for sprite", \
#format: #member, \
#range: suitableMembersList, \
#default: theMember \
], \
#myRolloverMember: \
[ \
#comment: "Rollover member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 1) \
], \
#myMouseDownMember: \
[ \
#comment: "MouseDown member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 2) \
], \
#myDisabledMember: \
[ \
#comment: "Disabled member", \
#format: #member, \
#range: suitableMembersList, \
#default: member (theMemberNum + 3) \
] \
] \
]

  -- Check if any sounds are available
  soundsAvailable = GetSuitableMembers (me, [#sound])
  if soundsAvailable.count() then
    soundsAvailable.addAt(1, #none)
    descriptionList.addProp \
(\
#myMouseDownSound, \
[ \
#comment: "- SOUNDS - Sound to play on mouseDown", \
#format: #sound, \

```

```

#range: soundsAvailable, \
#default: member (theMemberNum + 4) \
]\
)

descriptionList.addProp \
(
#myMouseUpSound, \
[ \
#comment: "Sound to play on mouseUp", \
#format: #sound, \
#range: soundsAvailable, \
#default: member (theMemberNum + 5) \
]\
)
end if

-- Place remaining properties at the end
descriptionList.addProp \
(
#myActiveFlag, \
[ \
#comment: "- INTERACTION - Button is initially", \
#format: #string, \
#range: ["Active", "Inactive"], \
#default: "Active" \
]\
)
descriptionList.addProp \
(
#myXRayFlag, \
[ \
#comment: "Sprites which cover the button", \
#format: #string, \
#range: ["block all mouse events", "let all mouse events through"], \
#default: FALSE \
]\
)
descriptionList.addProp \
(
#myMessageType, \
[ \
#comment: "Action on mouseUp", \
#format: #string, \
#range: ["do", "sendAllSprites", "call the actorList", "no action"], \
#default: "sendAllSprites" \
]\
)
descriptionList.addProp \
(
#myMessage, \
[ \
#comment: "", \
#format: #string, \
#default: "YourCustomMessage" \
]\
)

return descriptionList
end getPropertyDescriptionList

```

on GetSuitableMembers me, permittedTypes

-- Returns a list of all members in all casts

-- corresponding to any of the list of types

curs or 4

suitableMembersList = []

maxCastLib = the number of castLibs

repeat with theCastLib = 1 to maxCastLib

maxMember = the number of members of castLib theCastLib

```
repeat with memberNumber = 1 to maxMember
  theMember = member(memberNumber, theCastLib)
  if permittedTypes.getPos(theMember.type) then
    if theMember.name = EMPTY then
      suitableMembersList.append(theMember)
    else
      suitableMembersList.append(theMember.name)
    end if
  end if
end repeat
end repeat
cursor - 1
return suitableMembersList
end GetSuitableMembers
```

```
on PermittedMemberTypes me
  -- sent by:
  -- getBehaviorDescription
  -- isOKtoAttach

  return [#bitmap, #filmLoop, #flash, #movie, #picture, #quickTimeMedia, \
#shape, #vectorShape]
end PermittedMemberTypes
```