Building Dialogue - Documentation & XML Reference Manual

by William Chilcote

Contents

Intro to XML

If you already have any basic experience with Extensible Markup Language (XML), you should be able to skip to the next section. However, if you have never used XML before, this section will cover all the basics necessary for using the dialogue system.

Firstly, everything inside an XML document is considered a node. The correct syntax for a node with a *start tag* and *end tag* is as seen below.

```
<element> </element>
```

Node `element` is empty in the example above, but nodes can also contain other nodes within them.

```
<parent>
  <child/>
  <sibling>Some text</sibling>
</parent>
```

A node contained within another node is said to be a child of its parent node, as labelled above. Also note that node `child` does not have a distinct exit tag. This is because it is an *empty element*, as denoted by the forward slash toward the end. Two nodes with the same parent node are said to be sibling nodes. As seen in node `sibling`, nodes can not only contain other nodes within their start and end tags, but also generic text. Although not defined by any explicit tags, this is considered in XML to be a text node. Text nodes generally cannot be included as siblings to other nodes, the practice of which is known as *mixed content*. While this is done for text effects as explained later in this manual, this is makes use of a custom parsing solution.

Every XML document needs to have a single node that everything within the document is contained in. This node is sometimes referred to as the *root* node.

As well as containing child nodes, each node can carry its own information included in the start tag. This is done with *attributes*, as can be seen below.

```
<node attr1="Some value" attr2 = "Another value"> </node>
```

Attributes can be included on nodes with exit tags, as well as on empty elements. Any number of attributes can be included. The formatting around the = is whitespace insensitive. Attribute values can only contain strings, so any other data type, such as integers, will have to be parsed by the program.

Some characters, such as < and >, which are used as part of the node formatting, cause errors with the parsing if included as regular text. To prevent this, these characters each have a code, known as their *escape sequence*, that can be used instead, and will be rendered as the correct character by the program. These characters and their corresponding escape sequences can be found in the table below.

Symbol (name)	**Escape Sequence**
< (less-than)	&#60; or &lt;
> (greater-than)	&#62; or &gt;
& (ampersand)	&#38;
‘ (apostrophe or single quote)	&#39;
" (double-quote)	&#34;

Supported Nodes

Dialogue

The `dialogue` node is the root node for the XML document. All other nodes must be stored as children to this node, and it must not have any sibling or parent nodes.

```
<dialogue>
</dialogue>
```

Page & Say

The `page` and `say` nodes are where most of the dialogue is stored. These nodes should only be parents to a text node, and potentially text effect nodes as explained later. They do not make use of any unique attributes. Each `page` node represents a segment of dialogue that the object will display, before the user is required to press a button to continue to the next node, for example, another `page` node. `say` nodes are the same as `page` nodes, except that their contained text will be displayed as if it is being said by the player.

```
<page>This will be shown on a single page by the NPC.</page>
<say>This will be shown on a single page by the player.</say>
```

While user input is required to progress past each `page` or `page` node, the content of the node may also be split into more pages of text if overflow is detected, each additional page also requiring user interaction to progress. This is done automatically and requires no change in XML formatting.

Reply

The reply node is used to present the user with a set of responses to the dialogue host. The user is able to select one, which can change the direction of the dialogue. If it is determined that there are more available responses than can fit in the chat banner, they will automatically be split into multiple pages, with page navigation buttons included. The structure for a reply node is as follows:

```
<reply>
  <option>
    <say>This is the first option</say>
    <page>Response to the first option</page>
  </option>
  <option>
    <say>This is the second option</say>
  </option>
</reply>
```

Each selectable option in a reply must be stored as an option node, and as a child of the reply node. All option nodes must be siblings, and the reply node can have no other type of child nodes. The first child node of each option node must be of the say node type. This contains the text that will appear to the user as they select an option. This is a special case of the say node, as the contained text will not be displayed independently.

Upon selecting a particular option, the next node considered will be the node immediately after the say node. This could be, for example, a page node, another nested reply node, or nothing at all. If no nodes are placed after the say node, the next node considered is determined by methods explained in the Node Progression section below.

Goto

The goto node is used to redirect the current node elsewhere in the XML document. This is useful for providing multiple entrance points to the same segment of dialogue, or for creating loops. The destination node is defined by attaching a tag attribute, containing an identifying string to both the goto and the destination nodes.

```
<page>Please select an option.</page>
<reply>
  <option>
    <say>Option one.</say>
    <goto tag="Some tag"/>
  </option>
  <option>
    <say>Option two.</say>
    <page tag="Some tag">This is one of the best options!</page>
  </option>
  <option>
    <say>Option three.</say>
    <page>This is a terrible option.</page>
  </option>
</reply>
```

As seen in the above example, through the use of the goto node, the same dialogue can be displayed whether the user selects option one or two, without having to rewrite it.

Note that the goto node is an empty element, and requires no end tag. Be sure that the same tag string is not used between multiple destination nodes.

Finish

While the dialogue session will complete automatically upon reaching the final node, the `finish` node is used to manually terminate the session.

```
<reply>
  <option>
    <say>This option will allow the dialogue to terminate</say>
  </option>
  <option>
    <say>This option will immediately terminate the dialogue</
say>
    <finish/>
  </option>
  <option>
    <say>This option will show more dialogue</say>
    <goto tag="more"/>
  </option>
</reply>
<finish/>
<page tag="more">This is more dialogue.</page>
```

As the node is an empty element, it requires no end tag. This node can be useful if one particular dialogue response should cause the dialogue to terminate, or if additional nodes, reached through a `goto` node, are stored at the end of the file, after the main dialogue.

Random

The random node is used to randomly progress to the contents of one of the child option nodes.

```
<page>Watch me flip this coin.</page>
<random>
  <option weight="0.2">
    <page>Heads!</page>
  </option>
  <option>
    <page>Tails!</page>
  </option>
</random>
```

The optional weight attribute accepts a number between 0 and 1, representing the likelihood that the associated option will be selected. The compliment of the sum of all defined weight will be evenly distributed among the option nodes that do not have explicitly defined weights. Once selected, the current node moves to the first child of the option node.
In the example above, the first option has a 20% chance of being selected, while the second option has an 80% chance of being selected.

Escape

The escape node is used to set whether or not the user can manually exit the chat session with the escape key. The ability to escape a chat session is unique to each chat session, and is not stored between sessions. Whenever a chat session is initialized, the ability to escape it defaults to true. The escape node accepts an optional first attribute on that stores a bool. The ability to manually terminate this chat session with the escape key is set according to this bool. If the on attribute is omitted, the node will simply toggle the ability.

```
<escape on="true"/>
```

The escape node is self-closing, with no inner text. The node can be useful for dialogue that is critical to the storyline, for which the player should not be able to exit.

Call

The call node is a very powerful node that can be used to run custom methods from within dialogue. For a method to be eligible to be called with the call node, it must be public, part of a class that inherits from Chattable, and accept either no arguments, or a single string array. The method must also be part of the sample Chattable inheriting class that supplied the Dialog object to the ChatManager, as a reference to the class is passed to the Dialog constructor. A simple example of an acceptable method implementation is as follows.

```
public class ChattableInherited : Chattable {

    public void TestMethod (string[] args) {
        foreach (string item in args)
            Debug.Log(item);
    }
}
```

The required first attribute method must contain a caps-sensitive string that is the same as the name of the method to be called. call also accepts any number of attributes after method, each representing an addition to the string array that can be passed to the method. Due to the limitations of XML, each attribute must have a different name. The names of the string argument attributes are arbitrary however, as the only relevant values are those of the strings contained in the attributes. The only requirement is that the names all be unique, and are not "tag", as that is reserved for the goto node. If there are no additional attributes, the method will be run without the string array argument. To call the above example method, the implementation might look like the example below.

```
<call method="TestMethod" arg1="First" arg2="Second"/>
```

Set

Note: This node makes heavy use of variables. For more information, see the section 'Variables' below.

The Dialog object derived from a given XML document contains a set of stored properties of type Dictionary<string, bool>. The Dialog class also contains a static dictionary of a similar type, representing stored properties respective to the player, instead of a particular dialogue host. Properties in these dictionaries can be set using the set node. As public properties, these dictionaries can also be edited programmatically through other scripts. The set node accepts two attributes, var and value.

```
<page>Hello, who are you?</page>
<set var="N:Met Before" value="R:true"/>
```

The var attribute contains the string representing the dictionary key for the property. The data source for the variable in var must be of type N, P, or I, and defaults to type N. The value attribute defaults to type R. The value attribute is optional, and will result in setting the property to 1 or true if omitted.

Although a specific order of the two attributes is not required, it it generally recommended to include the var attribute first. The set node is an empty element and does not require an end tag. Property data is stored persistently between dialogue sessions.

Though not particularly useful on its own, the primary value of the set node is in setting variables to be utilized in the if node, as explained below, or manually read in a separate script.

If

Note: This node makes heavy use of variables. For more information, see the section ‘Variables’ below.

The if node is used to progress to a particular dialogue segment if a given condition is met, utilizing stored properties. The if node accepts up to three attributes, var, case, and value.

```
<if var="N:Met Before" case="E" value="R:true">
  <true>
    <page>Hey, I remember you!</page>
  </true>
  <false>
    <page>Hello, who are you?</page>
    <set var="N:Met Before" value="R:true"/>
  </false>
</if>
```

The case attribute contains a reference to the boolean operator to compare the two values in var and value with. If the case attribute is omitted, the operator type will default to ‘equal’. The strings for each operator are as follows.

Attribute string	Boolean operator	Operator (C)
E	Equal	=
NE	Not equal	!=
GT	Greater than	<
LT	Less than	>
GTE	Greater than or equal	<=
LTE	Less than or equal	<=

If both var and value can be cast to a bool or float, then all operators can be used. Otherwise, only the operators E and NE can be used.

The current node will move to the contents of the child node, true, if the condition defined by var, case, and value, is evaluated as true. If it is evaluated as false however, the current node will move to the contents of the child node false. Similarly to the set node, if the value attribute is omitted, it will default to 1 or true. If the correct destination node, being either true or false, is omitted, the current node will progress past the if node. If the property defined by var has not been set in its appropriate data set, the property will be considered false. var defaults to type N, and value defaults to type R.

Give & Take

Note: These nodes make heavy use of the Inventory System. For more information, see the included documentation, 'Item Creation & Modifying Player Inventory'.

The `give` and `take` nodes are both very similar in that they are able to change the amount of a given item in the player's inventory. The only difference is that `give` will add the amount to their inventory, while `take` will remove it. Both node types require the attribute `item`, the value of which should match the unique name of the item to be added or removed. They also both support an optional second node, `count`, which contains an integer representing how many of the given item should be added or removed. If the `count` node is omitted, the amount will default to `1`.

```
<page>Here, have a notebook!</page>
<give item="Notebook"/>
<reply>
  <option>
    <say>Thanks! I'm going to go write stuff in it!</say>
  </option>
  <option>
    <say>You can keep it, I already have one!.</say>
    <take item="Notebook" count="999999"/>
  </option>
</reply>
```

Advance

Note: This node makes heavy use of the Task System. For more information, see the included documentation, ‘Building Player Tasks’.

The `Advance` node interfaces with the Task System. It is a simple self-exiting node with no required or optional attributes. The purpose of the `Advance` node is to increment the current progress on the currently active `Step` by `1`. If the currently active task is of type `dialogue`, calling `Advance` will complete the entire step. Although this node will increment the progress of any `Step` type, it is intended for `dialogue` steps.

The following sample dialogue will progress the player to their next step, provided their current task is “Talking” from level “Intro”.

```
<dialog>
  <page>Keep talking to me to finish this task!</page>
  <if var="T:current" case="E" value=“T:Intro/Talking/0">
    <true>
      <advance/>
      <page>You completed this task!</page>
    </true>
  </if>
</dialog>
```

Variables

Certain node types in the Dialogue System make use of variables. These variables are referenced by the nodes from strings stored in their attributes. See the above node-specific documentation for which attributes accept variables.

As there are multiple sources of data in the game that a variable could be referencing, each variable's identifying string must start with a capital letter, representing the data source, followed by a colon, and then the identifying information from within that data set. The supported data sets are as follows.

Data Set Character	Description	Data Identifier Within Set
R	Raw input	Any string - can be parsed to bool, float, or int
N	Reference entries in the <string, string> dictionary associated with this NPC or Chattable object	Key (string) for the desired value in the dictionary associated with this NPC or Chattable object
P	Reference entries in the <string, string> dictionary associated with the player	Key (string) for the desired value in the dictionary associated with the player
I	Count of an item in the player's inventory	Name of an item
T	Chronologically ordered index of a step	Name of level/Name of task/0-based index of step - 'current' will return the index of the current step. Task and/or step information can be omitted to return the index of the first subtype in the enclosing level or task.

Each attribute that accepts a variable type has a default type. If the character identifying the data set is omitted, the system will assume the variable is in the default set as defined by the attribute. See the documentation for each variable-accepting attribute for their default types. It may be a good idea always include the data set character regardless of defaults in order to avoid ambiguity in dialogue formatting.

Though on the surface, variable support includes both boolean and integer types, everything is stored as a string. During comparisons or assignment, strings will be parsed to booleans or floating point values if and when possible. Similar to C++, boolean values are stored as implicit integers, with `0` representing `false`, and `1` representing `true`.

For example, `"I:Notebook"` will return the integer count of notebooks that the player has, and `"R:5"` will simply return `5`. `"Level1/Task1/2"` would return the second step from the task named "Task1" in level "Level1".

Variables can also be used in the inner text of other nodes to dynamically display text in the game. The name of the node must be the Data Set Character, and the first and only attribute must be `ID`, and contain the Data Identifier Within Set. This can be useful in nodes such as `page`, or `say`. The example below would display the number of pencils that the player has.

```
<say>I have <I id="Pencil"/> pencils with me.</say>
```

In-text variables can also be used while nested within Text Effects.

Node Progression

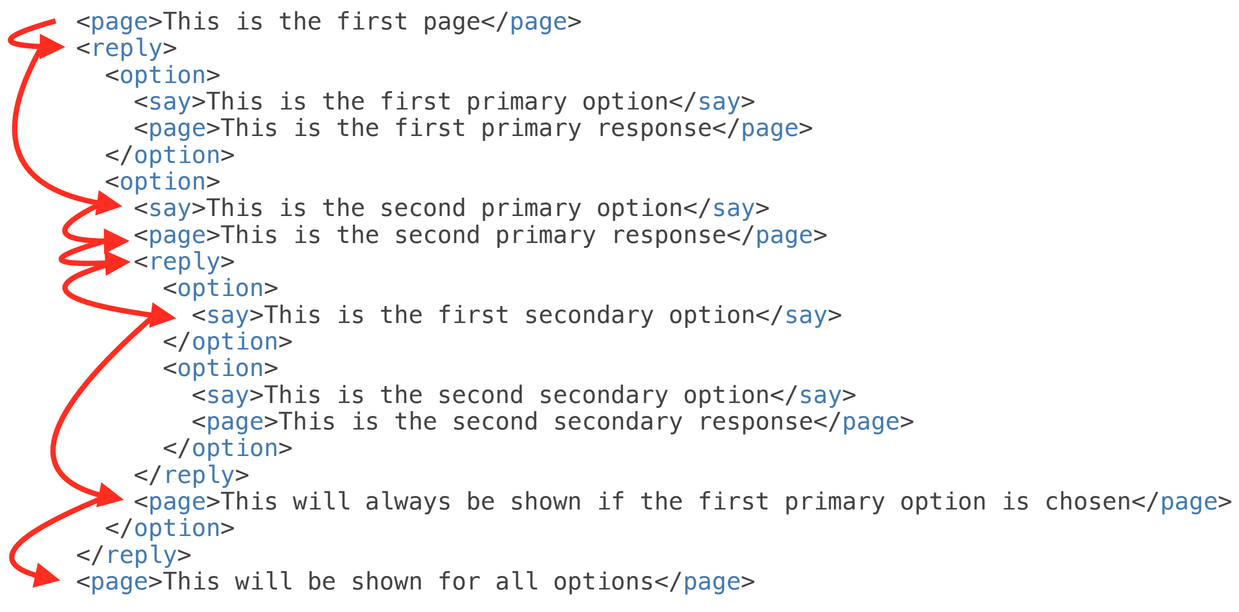
At all times during the progression of a dialogue instance, there is a particular node that the program is focused on. Excluding the use of `goto` nodes, this node moves linearly, from one node to its immediate sibling.

There are two nodes that the current node can be set at between periods of user input - `page` and `reply`. Upon receiving input from the user, if the current node is of type `page`, the current node will simply progress to the next node. However, if the current node is of type `reply`, the current node will be set to the child `say` node of the selected `option` node before progressing to the next node.

There are also *redirect* nodes. These include `random`, `goto`, `set`, and `if`. If the current node is ever set to one of these node types, the node instructions will be processed, and the current node will immediately be redirected, as defined by the type of redirect node. In the event that there are multiple consecutive redirect nodes, this is done recursively.

Upon reaching the final sibling node, in order to keep progressing, the current node must move up two node levels, and then to the next sibling of that parent node. It ascends two node levels in order to avoid node types such as `option` and `true`, which were rejected in favor of one of their sibling nodes. It will do this until finding a node with an available sibling, or until reaching the root node. If the current node is able to reach the root node before finding another node to progress to, the dialogue instance will be automatically terminated.

An example of a typical node progression sequence can be seen below. This example assumes that the user selects the second primary option, and then the first secondary option.



```
<page>This is the first page</page>
<reply>
  <option>
    <say>This is the first primary option</say>
    <page>This is the first primary response</page>
  </option>
  <option>
    <say>This is the second primary option</say>
    <page>This is the second primary response</page>
    <reply>
      <option>
        <say>This is the first secondary option</say>
      </option>
      <option>
        <say>This is the second secondary option</say>
        <page>This is the second secondary response</page>
      </option>
    </reply>
    <page>This will always be shown if the first primary option is chosen</page>
  </option>
</reply>
<page>This will be shown for all options</page>
```

Text Effects

In addition to the node types used to organize the dialogue, certain nodes can also be used within the dialogue itself for formatting purposes. All text effect nodes must have a start tag, end tag, and a portion of dialogue inside them. Most effects also make use of attributes for further customization. Text effect nodes can also be nested to apply multiple effects. If multiple text effect nodes of the same type are nested however, the properties of the innermost node will override the others for that segment.

The process of adding support for new text effect nodes is relatively simple, as each node type has its own class, independent from the rest of the dialogue system. Each class must inherit from the `TextEffect` class, and override several methods. The public property `nodeName` must be specified in the constructor, and contain a string representing the name of the node as it is denoted in the XML document. The public method `SetProperties` must be overridden, and should contain the logic responsible for interpreting the relevant information from the given `XmlNode`. A second public method, `ApplyEffect` must also be overridden. This method should contain the logic responsible for applying the relevant effect to the given `Text` object. Once the child class has been written, add a new instance of it to `TextStyle.allEffects`, as defined in its constructor, and the effect will start being applied to dialogue.

The class for `TextColor` is provided below to serve as an example of implementation.

```
public class TextColor : TextEffect {

      public Color color = Color.white;

      public TextColor () {
            nodeName = "color";
      }

      public override void SetProperties (XmlNode input) {
            string hex = input.Attributes.GetNamedItem ("hex").Value;
            color = HEX2RGB (hex);
      }

      public override void ApplyEffect (Text input) {
            input.color = color;
      }

      Color HEX2RGB (string hex) {
            // This logic is not necessary for this example
      }
}
```

TextColor

Node name:

color

Attributes:

1. hex: Hexadecimal RGB value of a color at full opacity

Description:

The TextColor effect is used to make the given portion of dialogue render in any given color.

Example:

Display the sentence “I am an NPC.” with the word “NPC” rendering as blue.

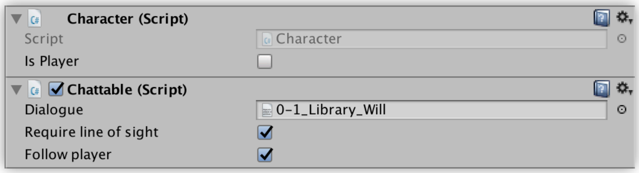
```
I am an <color hex="0000FF">NPC</color>.
```

Creating Interactable Objects

Once the dialogue XML document has been written, following the methods explained above, only a few more steps remain before being able to see it rendered in the game. After importing the document into Unity as a `TextAsset`, locate the `GameObject` with an instance of the `Chattable` class attached. In the Inspector, populate the field for `dialogData`, “Dialogue,” with the imported `TextAsset`. Once this property is no longer null, the object will become interactable upon approaching. A `Dialog` object is automatically generated from the given `TextAsset` in `Chattable.Start()`. The dialogue `TextAsset` can be set programmatically to allow for a change in dialogue information at runtime. This is done using `Chattable.ChangeDialog()`, which will also update the derived `Dialog` object.

The checkbox for the boolean property `requireLineOfSight`, “Require line of sight,” is enabled by default, and requires there to be a direct path, unobstructed by non-trigger colliders between the player and this object for dialogue to be initiated. This can be used to assure that objects cannot be interacted with through walls. However, if an NPC is behind a collider-enabled desk for example, this checkbox could be disabled.

The Inspector view of the `Chattable` component changes depending on whether or not an instance of `CharacterCreation.Character` is also attached to the `GameObject`. If an instance of `CharacterCreation.Character` is attached, the only additional visibly serialized property will be `followPlayer`. If you would like this character to look at the player when they are in range, check the “Follow Player” checkbox. If this is not checked, the character will not change their direction when the player gets close. Sprite information from the character customizations will be used automatically in chat banners, similar to the way that the player’s customizations are automatically displayed in chat banners for `say` and `reply` nodes.

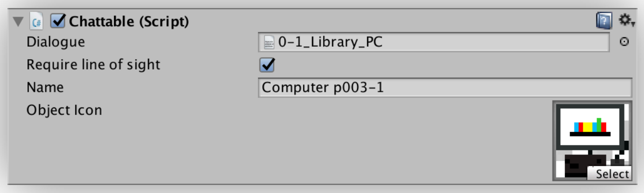

Character (Script)

Script	Character
Is Player	☐

Chattable (Script)

Dialogue	0-1_Library_Will
Require line of sight	✓
Follow player	✓

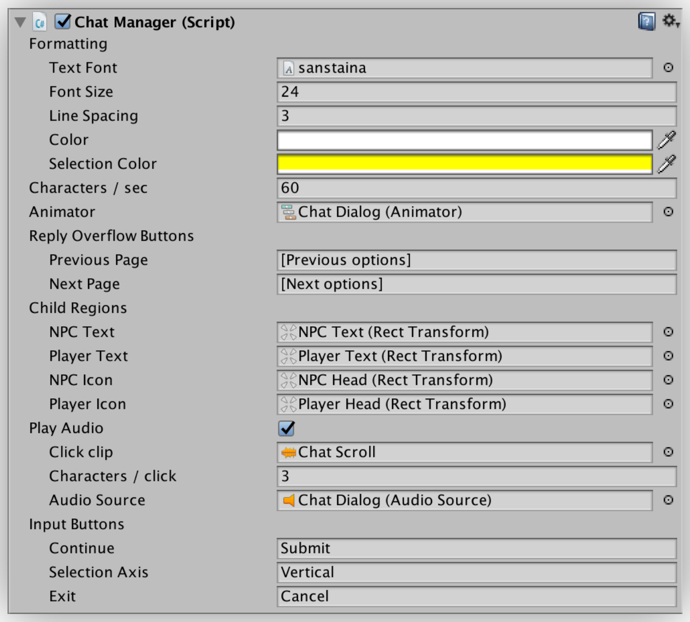
If an instance of `CharacterCreation.Character` is not attached, two additional fields will be serialized in the Inspector - a sprite that will be used in the chat banner to represent the host object, and a string containing the name of the object as will be shown on the banner. If the sprite is null, this information will not be shown.


Chattable (Script)

Dialogue	0-1_Library_PC
Require line of sight	✓
Name	Computer p003-1
Object Icon	

System Setup

Global properties, provided for further customization of the dialogue's appearance, are available through serialization on the singleton instance of the `ChatManager`, attached to the root `RectTransform` of the chat banner. These properties include the font, font size, spacing in pixels between lines, default text color, color of the currently selected response, and characters per second for scrolling text. Several `RectTransform` components are serialized as well, representing the regions that the dialogue and icons are confined to for object and player dialogue, typically children of the `RectTransform` banner that the `ChatManager` is attached to. `ChatManager` also makes use of properties that define how audio will play. As chat scrolls, it is possible to repeatedly play an audio clip. The given clip will play whenever the given number of characters are rendered, on the given `AudioSource`. There are also strings representing the names of inputs as defined in `InputManager`. All of these properties are dynamically serialized in the Unity Inspector through a custom editor.



Chat Manager (Script)

Formatting	
Text Font	sanstaina
Font Size	24
Line Spacing	3
Color	⬜
Selection Color	🟨
Characters / sec	60
Animator	Chat Dialog (Animator)
Reply Overflow Buttons	
Previous Page	[Previous options]
Next Page	[Next options]
Child Regions	
NPC Text	NPC Text (Rect Transform)
Player Text	Player Text (Rect Transform)
NPC Icon	NPC Head (Rect Transform)
Player Icon	Player Head (Rect Transform)
Play Audio	✓
Click clip	Chat Scroll
Characters / click	3
Audio Source	Chat Dialog (Audio Source)
Input Buttons	
Continue	Submit
Selection Axis	Vertical
Exit	Cancel

The core dialogue system is made up of three different scripts. An instance of `Dialog` is constructed from a given XML document, and is responsible for setting its public properties to store information contained in the current node. It has two public methods, `Next` and `ToBeginning`. The public property `selectedResponse` is used when `Next` is called on a `reply` node to progress to the appropriate option. `Dialog` does not inherit from `MonoBehaviour`, so it is not attached to a `GameObject`. Instead, an instance of it is kept on `ChatManager`. `ChatManager` is responsible for reacting to user input, and making the appropriate changes to the `Dialog` instance in use. It then reads the updated properties from `Dialog` and updates the chat interface. As `ChatManager` only processes inputs and outputs to and from `Dialog`, it would be possible to write a custom alternative, without having to make any modifications to `Dialog`. The third script, `TextStyle.cs`, also contains `TextEffect` and several derived classes in the same file. None of these classes inherit from `MonoBehaviour`, and are used by `ChatManager` for displaying styled text.