# Table of Contents

1: Introduction...................................................................................... 5  
- Overview .......................................................................................... 5  
- File Structure .................................................................................. 6  
- Editing Files ................................................................................... 6  
- Command Basics .......................................................................... 7  
- Support .......................................................................................... 11  

2: Scalar Processing........................................................................... 12  
- Overview .......................................................................................... 12  
- Changing Case ............................................................................... 13  
- Numeric Operations ....................................................................... 14  
- Joining, Splitting and Trimming ..................................................... 14  
- Special Characters ......................................................................... 16  
- Grammar ......................................................................................... 18  

3: List Processing ............................................................................... 20  
- Overview .......................................................................................... 20  
- Creating Lists ................................................................................ 21  
- Loop Processing ............................................................................ 23  
- Working with Parts of a List ......................................................... 25  
- Re-Sorting a List ............................................................................ 28  
- Lists vs. Hashes ............................................................................ 30  

4: Conditional Processing ................................................................. 33  

5: Flow Control .................................................................................. 36  
- Advancing the Loop ...................................................................... 36  
- Sleeping and Blocking .................................................................. 36  
- Pausing and Stopping ..................................................................... 38  

6: Date/Time/Temp Functions ............................................................. 40  
- Advanced Functions ....................................................................... 41  

7: Global Variables ............................................................................ 44  

8: Environment Information ............................................................... 46  
- Runlevel Information .................................................................... 46  
- Output Device ............................................................................... 47
9: Advanced Functions ................................................................. 49

10: Data Elements & Filters ............................................................. 65
   Closings ....................................................................................... 66
   Currents ....................................................................................... 72
   Elections ....................................................................................... 73
   Finance ......................................................................................... 76
   Forecasts ....................................................................................... 77
   Headlines ..................................................................................... 78
   Lottery ......................................................................................... 79
   SevereWx ................................................................................... 79
   Ski ............................................................................................... 89
   Sports ............................................................................................ 90
   Terrorism .................................................................................... 94
   TrafficMan thos............................................................................ 94

11: LOGIC Cookbook ....................................................................... 97
   A. Presentation Techniques and LOGIC-Writing Tips............. 97
   B. Responding to Real-Time Data Values............................ 102
   C. Sorting, Filtering and Grouping ........................................ 110
   D. Working with Sponsor Logos ............................................. 116
Overview

LOGIC is NewsTicker’s powerful automation language. It is the middleware between the automated and manually entered data in NewsTicker’s database, and the various destinations to which you want the data sent.

At the core, NewsTicker is a content management system. Data is stored in a database unformatted. When it comes time to actually use the data, that’s when the formatting takes place. This allows you to present the information in an endless number of ways on-air, on the web, even on a scrolling LED sign.

To call LOGIC a formatting language would only be touching on the surface of what it can do. It’s so powerful, it can allow you to establish real-time decision-making trees so that NewsTicker will follow a prescribed action sequence. With it, you could rotate sponsors every third sports score, use a “cold weather” sponsor if the current temperature drops below a certain threshold, or automatically remove all other content modules when the number of active school closings meets your specific requirement.

Note  We recommend you first familiarize yourself with NewsTicker by reading the NewsTicker User’s Guide and any of the specialty module documentation as it will familiarize yourself with other conventions used in this book.
File Structure
LOGIC files are called for within runlevels. Each module within a
runlevel must be associated with a respective LOGIC file, although
multiple instances of a single module can be associated with different
files. Since you can have an unlimited number of runlevels, you can also
have an unlimited number of LOGIC files to drive them.

LOGIC files reside on the NewsTicker server. In the /web/htdocs/formats
directory are sub-directories named for each module, such as Currents
and Headlines, where all LOGIC files are stored. So, all automation files
for the Sports module are located in /web/htdocs/formats/Sports.

In addition, there are special files that run once at the beginning of a
runlevel. These initialization LOGIC files are stored in the
/web/htdocs/formats/Initialization directory.

To be properly recognized by NewsTicker as a LOGIC file, be sure to
save it using the .fmt extension.

Editing Files
LOGIC files are simply text files that use the .fmt extension. Any text
editor on your PC can be used to create and modify the file, such as
WordPad. You can then use an FTP client to transfer the file to and
from the NewsTicker server.

Note  NewsTicker Support may need to configure permissions on your
server to write into the formats directory.

To facilitate editing LOGIC files, you may wish to invest in a
commercial text editor that is capable of opening and saving directly to
an FTP server, such as UltraEdit (or UEStudio) for Windows or BBEdit
for MacOS.

Using the LOGIC Editor
Built into the Broadcast module is a rudimentary text editor designed
specifically for editing LOGIC files. From the Broadcast module, click
LOGIC Editor. From there, you can navigate into module-based
directories and edit your files.

Using Pico
Pico is a UNIX-based text editor that resides on your NewsTicker
server. (Other advanced editors exist too, such as vi and emacs.) By
using Pico, you can edit LOGIC files without FTPing since you are
working directly on the server.
To start the editor, ssh into the NewsTicker server and change into the directory where you wish to create or edit a file. Then, type:

```
pico filename.fmt
```

If the file already exists, then Pico will open the file for editing. If it doesn’t exist, it will be created as a new file.

When Pico loads, you can begin typing your LOGIC file. You can move around the file using the arrow keys. Along the bottom, you’ll see a list of commands, each preceded with the ^ symbol. The ^ symbol denotes the Ctrl key. So, ^G for “Get Help” is accessed by typing the Ctrl-G key combination.

There are two ways to save your work. To save without exiting the file, which is useful for making a quick change and testing or for writing to disk what you’ve done so far, use the Ctrl-O (oh) command. Pico will suggest a filename using the current name as a default. Simply press Enter to save using the existing filename. To exit, use Ctrl-X. If changes have been made to the file, you will be asked if you would like to save the changes first. Type N to exit without saving your changes. Type Y followed by Enter to save them.

**Additional editing commands:**

```
Ctrl-A           Move cursor to beginning of line
Ctrl-E           Move cursor to end of line
Ctrl-V           Move cursor forward a page of text
Ctrl-Y           Move cursor backward a page of text
Ctrl-^           Mark the current cursor position as the start of a text selection (for copying or cutting)
Ctrl-K           Cut (delete) the current line or marked text
Ctrl-U           Uncut (paste) the text that was last deleted, at the current cursor position. By cutting in one location, moving the cursor to a new location, and then uncutting, you can move text easily.
Ctrl-L           Redraw the screen
Ctrl-G           Get Help (Display the Pico help file)
```

For more in-depth tutorials, search the web for pico editor tutorial.

**Command Basics**

LOGIC files are a combination of output device commands and LOGIC commands. A character generator is the most common output device for NewsTicker, and CG-specific commands include those to change into a directory, write a page of data, display the page, etc. LOGIC-
specific commands do things like iterate over a list of all headlines and put a crawl separator between each, or detect when a sports team has won to turn it a different color.

Within a LOGIC file, the most basic layout involves doing a series of actions at the beginning of the file, then a series of actions based on the data in the module, and finally a series of actions after the data in the module has been shown.

For example, let’s assume you want to show lottery results. In the LOGIC file, you tell NewsTicker to loop through all of the games and give instructions on how to display them. If there are five lottery games, you do not need to tell NewsTicker how to display each one individually. Instead, you create what is called a FOREACH loop, since what you are saying is “for each game, do this...” Then, within that loop, you give generic instructions on where the name of the lottery game should go, where the results should go, where it should save the data, instructions on how to show the data, and then a command to tell it to sleep for a few seconds. If there are more games to show, NewsTicker will automatically loop back up to the beginning of the FOREACH loop and rerun your set of commands using the information for the next lottery game.

```
\m\c:/Templates/Production\[\% CRLF %
\[\% FOREACH game = Lottery \%
\[\% \Lotto\[\% game.game_name \%
\[\% \game.results \%
\[\% CRLF \%
\[\% SLEEP(3) \%
\[\% END \%
\]
\[\Lotto\More results on www.bignews.com\[\% CRLF %
\[\% SLEEP(3) \%
```

LOGIC directives are embedded within the character sequences [% and %]. Anything between those will be interpreted by the broadcast engine as a LOGIC command. Things not between those character sequences are destined for the output device (CG in this case).

The FOREACH command starts the loop. In this case, it could be read in human form as “For each game listed in the Lottery module, do the following.” NewsTicker will do just that until it sees [% END %], which is NewsTicker’s cue to move to the next lottery game if there is another one and redo the code between the FOREACH and END. If not, it will do whatever is next in the LOGIC file, if anything.
Note that before the `FOREACH` command, it tells the CG to set the message directory using the `M` automation command. This command is CG-specific but is common among devices that emulate Chyron’s Intelligent Interface protocol. Next, it goes through each game and writes a page to the CG using the `w` command, displays it using the `Y` command, and then waits a couple of seconds before displaying the next game. Once they’ve all been displayed, it finally shows the station’s website address.

Most NewsTicker LOGIC files will contain a `FOREACH` statement. Its very name implies what it does – it takes a list and loops through each item in the list and does something with it. In our example, Lottery is the name of the list (conveniently named the same as the module) and it has as many items in the list as there are games to show results for.

An important concept to understand is each item in the list has multiple attributes. Sports scores are the easiest way to visualize this. Each item in the `Sports` list is a single game. Of course, for each game, there is a league name, home team, home score, visiting team, visiting score, status, etc. Those are the various attributes. Since you are in control of your own LOGIC file, you get to specify the exact attributes you want to display. So, in the `FOREACH` loop, you set a temporary name for the current game in the loop. In our example, that temporary name is `game`, but it could be anything you like that is syntactically correct. Then, you can specify the attribute you want by using dot-notation where the left side of the dot is the temporary name and the right side is the attribute name, as in `game.results`.

**Note** A complete list of attributes for each NewsTicker data module is provided in Chapter 10. For advanced programmers, it may be easier to think of Lottery above as being an ordered array where each item in the array contains an associative array – put another way, an array of hashes.

The above example is clearly simplified since one of the big features that sets NewsTicker apart from its competitors is LOGIC’s ability to perform arbitrary on-the-fly decision making at the time of broadcast. In fact, NewsTicker makes very few assumptions about how you will want to show the information that is within the database. This allows stations to completely customize the on-air appearance and corresponding automation without any NewsTicker software changes.

But even as your LOGIC files get more complex, they are all virtually constructed the same way – using a `FOREACH` loop to present the main content portion of the module, and placing items before or after the loop to precede or follow the content.
Comments
When writing LOGIC files, you may wish to annotate your work, or temporarily disable a section of code. Comments allow you to do this. All comments are ignored by NewsTicker and processing continues as normal after the comment is closed.

Enclose comments within [%# and %]. For example:

```logic
[%# Lottery LOGIC file, edited 5/1/05
No sponsor/Morning show %]
[% FOREACH game = Lottery %
\W\1\Lotto\ [% game.game_name %]
\[% game.results %\][% CRLF %]
\[% CRLF %]
\[% CRLF %]
\[% SLEEP(3) %]
[% END %]
```

In the above example, the highlighted code is not processed by NewsTicker since it is “commented out”.

Log Messages
There are times when debugging your LOGIC files that you may wish to send messages to the server’s log file. This is especially useful if you want to physically see the value of a particular data field or operation without sending it to the CG.

To do this, use the | stderr notation. The string preceding the pipe character (|) will be sent into the log file, as in:

```logic
[% x = 0 %]
[% FOREACH game = Lottery %
\[% 'The value of x is ' _ x _ '\n' | stderr %]
\W\1\Lotto\ [% game.game_name %]
\[% game.results %\][% CRLF %]
\[% CRLF %]
\[% CRLF %]
\[% SLEEP(3) %]
[% END %]
```

The log file would say `The value of x is 0` on the first loop, and increment from there. The \n notation simply inserts a new line character (carriage return).

If a runlevel is started from the web GUI, /var/log/httpd/error_log is used. If it is started from the intelliCommander, /web/logs/ic_air_log is used instead.
Support
As with all of NewsTicker, support is available by emailing support@newsroomsolutions.com or calling (336) 784-0686.
Overview

In scripting a LOGIC file, there are three types of data containers that may be used to store and manipulate content: scalars, lists, and hashes. The most simple data type is a scalar, sometimes referred to as a string.

A scalar variable stores a single piece of information – either a string or a number. The value of a scalar is inserted into the LOGIC file by using the variable’s name, such as:

```
[% sponsor %]
```

Since the value of a scalar wouldn’t be useful without a way to retrieve it, every variable has two parts: a name and a value.

```
[% sponsor = 'Big City Pools' %]
```

```
Today's news is brought to you by [% sponsor %].
```

In the above example, `sponsor` is the name of the variable and `Big City Pools` is currently the value. (You can change the value at any time.) As shown, you “use” the `value` by calling for it by its `name`.

This chapter deals with various ways in which you may manipulate a scalar, such as converting it to all uppercase or rounding a number. The next chapter will describe in more detail the complex variables lists and hashes, but you’ll note they both are a collection of multiple scalars, differing only in how they are accessed from within the group. Lists
maintain order so they are “keyed” on their position. Hashes don’t maintain order and are accessed by using the scalar’s name.

**Changing Case**

As with most of the commands in this book, the STRING can be a physical word or phrase entered directly into the LOGIC file surrounded by single or double quotes:

```latex
[\% \text{LC(‘Make This Headline Lowercase!’)} \%]
```

Or, they can be the name of the variable, as described earlier, without any quotes:

```latex
[\% \text{LC(sponsor)} \%]
```

The above example would send the value of the sponsor scalar to the CG (or other output device). If you wish to execute the command but wait to use it, you can capture the result back into the variable, as in:

```latex
[\% \text{sponsor = LC(sponsor)} \%]
```

Do some other things here...

```latex
[\% \text{sponsor} \%]
```

This is true of most LOGIC commands.

**LC**

```latex
\text{LC(STRING)}
```

Returns STRING in all lowercase letters.

**LCFIRST**

```latex
\text{LCFIRST(STRING)}
```

Returns STRING with the first letter lowercased.

**UC**

```latex
\text{UC(STRING)}
```

Returns STRING in all uppercase letters.

**UCFIRST**

```latex
\text{UCFIRST(STRING)}
```

Returns STRING with the first letter uppercased.

**UCWORDS**

```latex
\text{UCWORDS(STRING)}
```

Returns STRING with the first letter of each word uppercased. For example:
The above would print Here Are The Latest Headlines...

**Numeric Operations**

General arithmetic can be performed using + (add), - (subtract), * (multiply), / (divide), and % (modulo). The most common use of arithmetic is in advancing an internal counter, such as:

```
[mycounter = mycounter + 1]
```

The * and / work exactly as you would expect, multiplying or dividing their two operands. Division is done in floating point, unless you’ve converted the numbers to integers using INT. The % operator converts its operands to integers before finding the remainder according to integer division.

**COMMIFY**

```COMMIFY(NUMBER)```

This will return NUMBER with commas. For instance, if the volume of a stock is 2581230, the result of COMMIFY is 2,581,230.

**INT**

```INT(NUMBER)```

Returns the integer portion of NUMBER. For instance, `[INT(3.99)]` returns 3. You shouldn’t use this for rounding numbers because it truncates towards 0.

**ROUND**

```ROUND(PRECISION, NUMBER)```

Rounds a floating point number, NUMBER, to the number of decimal places specified in PRECISION. To round to a whole number, set PRECISION to 0. Use 1 for tenths and 2 for hundredths, etc.

**length**

```STRING.length```

This will return the number of characters in the given STRING.

**Joining, Splitting and Trimming**

**Joining Strings**

Joining multiple strings together can be useful, especially for repetitive operations. To perform a concatenation, use the _ operator, as in:

```data = stock.changeUpDown _ stock.changeAmount```
... or ...

```ruby
[% name = candidate.last_name _, ' ', candidate.first_name %]
```

Note that you must include white space around the _ operator and that it can include dynamic variables (like `candidate.last_name`) and static text (like the comma with a space).

**repeat**

```ruby
STRING.repeat(NUMBER)
```

This will repeat `STRING`, `NUMBER` number of times.

**Splitting Strings**

There are two basic ways to split strings – either by a specific character or by a designated length.

**split**

```ruby
STRING.split(PATTERN)
```

Using the given `PATTERN`, it will split `STRING` into as many parts as possible. `PATTERN` can be any legitimate regular expression. For instance, `[% numbers = game.results.split('-') %]` creates an array called `numbers` where each slice of the array is a lottery number. With this, you could put each number into its own graphical ball instead of showing the usual results separated by hyphens.

Since `PATTERN` can also be a regular expression, you could split on more than one character. See *Chapter 9: Advanced Functions, Match/Replace*, and consult documentation on regular expressions for more details.

**WRAP**

```ruby
WRAP(STRING, LENGTH)
```

Returns `STRING` broken into multiple lines where no line exceeds `LENGTH` number of characters. This is useful for older character generators that aren’t capable of word-wrapping. `WRAP` will not hyphen words – it will break on spaces. When assigning the output of `WRAP` to a variable, it will produce an array where each slice of the array is a new line.

**WRAPWIDOW**

```ruby
WRAPWIDOW(STRING, LENGTH)
```

This command works just like `WRAP` except if the last line would result in only a single word, commonly referred to in publishing as a widow, then it will move the last word of the previous line down too. This avoids a widow by creating a last line that contains two words.

**Trimming Strings**

Depending on the justification of your text objects, you may need to trim excessive white space on either side of the text string.
**LTRIM**

LTRIM(STRING)

Returns STRING with leading white space removed.

**RTRIM**

RTRIM(STRING)

Returns STRING with trailing white space removed.

Or, you can trim actual characters from the end of a string if you want to make sure a specific field is never longer than a specified number of characters.

**substr**

STRING.substr(OFFSET, LENGTH)

This command returns a substring starting at OFFSET for LENGTH number of characters. For example, [% game.forced_home_team.substr(0, 3) %] will return the first three letters of the home team’s name.

---

**Special Characters**

To allow you to format your code for readability, there are two types of characters that are scrubbed from all lines of all LOGIC files prior to parsing: (1) leading and trailing spaces, and (2) carriage returns.

**SPACE**

SPACE()

When parsed, all spaces are removed between the last character of the previous line and the first non-space character of the next line. This allows you to indent your LOGIC file commands for readability without physically sending those indentations to the output device. Using the SPACE command will force a space at a spot where it would otherwise be removed by the parser.

This command is especially useful when you want to make sure the lack of content from a data element won’t prematurely end a command sequence. For example, take this snippet of code:

```plaintext
w\_pg\template\%
[\% headline.display_line1 %]\n[\% headline.display_line2 %]\n[\% CLOCKTIME %]\n```

Here might be the command that goes to the CG:

```plaintext
w\_pg\template\Line 1 data\Line 2 data\8:03\n```
But, assume for a second that the producer didn’t enter a second line of text into the headline – that it is just the first line. The way the command is written, it would be sent to the CG as follows:

```
\w\_pg\template\Line 1 data\8:03\\n```

Note the double backslashes before the time. Since two backslashes is typically part of the “end of command sequence” for CGs, the command will have terminated before it sent the time.

You could change the LOGIC command to include a regular space, as in:

```
[\% headline.display_line2 %] \n```

But, since there is no second headline, the line will end up being a hard space and a backslash. The parser will remove the leading space thinking it is formatting indentation and you will be in the same situation as you started. The solution is the SPACE command:

```
[\% headline.display_line2 %][\% SPACE %]\n```

Now, the command sent to the CG will be:

```
\w\_pg\template\Line 1 data \8:03\\n```

You can also use the short circuit operator || (or), described in Chapter 4, to display a space only if the headline does not exist:

```
[\% headline.display_line2 || SPACE %]\n```

In the above example, the number zero would test false. So if you had a score of zero, it would send a space instead of the zero. To make the command safe for a numeric value of zero, you should test for the number of characters in the scalar instead of the value:

```
[\% game.vt_score.length || SPACE %]\n```

**CRLF**

Most character generators use two backslashes, a carriage return, and a linefeed as the command terminator sequence. Sending either of those before you wish to end the automation command will terminate the sequence prematurely.

Similar to spaces, carriage returns are useful for formatting the LOGIC file for readability. Therefore, all carriage returns and linefeeds are removed from the file by NewsTicker’s pre-parser. To then specify the sending of a carriage return and linefeed, the CRLF command is used.

```
\w\_pg\template\%
[\% headline.display_line1 %]\n[\% headline.display_line2 %][\% SPACE %]\n[\% CLOCKTIME %][\% CRLF %]
```
Zeroes
A 0 (zero) on a line of its own should be avoided as it could interfere with the broadcast engine's post-parser. In programming parlance, zero is often used to represent false, as in the binary off state. When the post-parser looks to see if there is data on the line to send to the output device, and it encounters a zero by itself, it assumes there is no data and continues on.

To avoid this, put another character on the line, such as a backslash when dealing with character generator automation. Or, force a space using the `SPACE` command.

ASCII
ASCII(DECVAL)
Some special characters, such as control characters or symbols, are more easily entered by using the decimal equivalent. The `ASCII` command will return the decimal value `DECVAL` as a character. `DECVAL` must be a four-digit value, zero padded. For example, the following will produce the degree symbol ° in most fonts:
```
[% ASCII(0176) %]
```

ASCII2HEX
ASCII2HEX(STRING)
If given a `STRING`, this method will return the hex values of each string. This is used for LED signs that need to be sent hexadecimal values instead of normal ASCII.

Grammar
These functions are provided for convenience when grammatically formatting dynamic text.

PLURALIZE
PLURALIZE(NUMBER, SINGULAR, PLURAL, ZERO)
This command lets you pass in a numeric value, as well as options for what it should return based on the value of that number. It is ideal for making words plural at runtime.

It looks at the value of `NUMBER`. If `NUMBER` is one, then it will return the string passed into `SINGULAR`. If it is greater than one, it will return `PLURAL`. If `NUMBER` is zero, less than zero, or empty, it will return `ZERO` if present or `PLURAL` if `ZERO` is not present.

The following is a contrived example showing the number of sports games being played today:
Results, based on the number of games, would be as follows:

0: There are no games being played tonight.
1: There is 1 game being played tonight.
2: There are 2 games being played tonight.

VOWELPICK

VOWELPICK(STRING, IFVOWEL, IFCONS, APPEND)
To assist in building grammatically correct sentences using dynamic data, the VOWELPICK command looks at the first letter of STRING and returns IFVOWEL if the first letter is a vowel, or IFCONS is the first letter is a consonant.

The most useful case for this command is building a dynamic sentence about active severe weather:

[% VOWELPICK( condition, 'An', 'A', 'Y' %] is in effect until ...

If condition is “Tornado Watch”, it would print “A Tornado Watch is in effect until …” If the condition is “Inland Hurricane Watch”, it would print “An Inland Hurricane Watch is in effect until …”

If the optional argument APPEND is Y, then STRING is appended to IFVOWEL or IFCONS when the command is executed. So the above example could have been written more efficiently:

[% VOWELPICK( condition, 'An ', 'A ', 'Y' %] is in effect until ...

Note the space after the IFVOWEL and IFCONS strings above when APPEND is used.

While this command works for all severe weather conditions, it’s important to note it doesn’t completely follow the grammatical rule of when to use “a” versus “an”. The true test is whether the word has a consonant or vowel sound. Thus, “a” should be used before a consonant sound (a U.N. resolution) and “an” should be used before a vowel sound (an FDA directive).
Overview

If scalars are the most basic of programmatic elements, then lists are one step above. In their most basic form, lists are simply a set of scalars put together in a prescribed order such that they can be removed one by one. They can be thought of as containers.

As an example, imagine you have some 3” x 5” note cards in front of you. On each card, you enter a different headline. When you are done, you end up with 10 different cards, each with a different headline. In LOGIC terms, you have 10 scalars or strings. Now, take the cards and put them in the order you want them to appear on-air, perhaps editorially by importance, and staple them. In LOGIC terms, you have just created a list of 10 elements because you have (1) grouped them together, and (2) defined an order.

Obviously, the above example is a rather simplistic one. But if you keep in mind the basic concepts, you should be able to build on it as we get deeper into how lists are used in NewsTicker and LOGIC.

For those who have previous programming or scripting experience, a list is an array. As with any array, it is a set of sequentially indexed elements having the same data type. The array may simply contain a series of strings, or a more complex data structure (such as an associative array, also known as a hash).
Creating Lists

Most of the time, you will be dealing with existing lists in LOGIC. After all, the whole idea of the LOGIC automation layer is to take existing lists (headlines, sports scores, school closings) and merely format them for air. But, there are times when you may wish to create your own list and they are simple to do.

There is nothing inherent in the name of a list that tells LOGIC it is a list instead of a scalar. In fact, since scalars are far more prevalent than lists, LOGIC will assume you are always working with scalars unless you specifically tell it otherwise. The practice of telling LOGIC you wish to create a list is known in programming circles as initializing the list or array.

```%
[ % sponsors = [] %]
```

The special notation `[]` tells LOGIC that `sponsors` will be a list, not a scalar.

**push** To add something to the list, you “push” it in. Pushing always adds elements to the end of the list.

```%
[ % sponsors.push('chevy.tga') %]
[ % sponsors.push('ford.tga') %]
[ % sponsors.push(GLOBAL2) %]
```

So, the elements follow the order in which you add them.

**unshift** The same way `push` appends things to the end of a list, `unshift` prepends them to the start of a list. If `chevy.tga` is the first thing in the list, and `ford.tga` is the second, and you:

```%
[ % sponsors.unshift('chrysler.tga') %]
```

The list is now, in order: `chrysler.tga`, `chevy.tga`, `ford.tga`.

**reverse** You can reverse the entire array by using the `reverse` command.

```%
[ % sponsors = sponsors.reverse %]
```

The list is now, in order: `ford.tga`, `chevy.tga`, `chrysler.tga`.

**Populating at List Initialization**

You can also create a list all at once. The following command will initialize and generate the entire list at one time:

```%
[ % sponsors = ['chrysler.tga', 'chevy.tga', 'ford.tga'] %]
```

Or, if you prefer to format it for greater readability:

```%
[ % sponsors = [
   'chrysler.tga',
   'chevy.tga',
   'ford.tga'
]
```
Even if you populate the list at the time of creation, like the example above, you can continue to add elements by using `push` and `unshift`.

You can use this same notion of (re-)initializing and populating at once to empty your list after you’ve put values into it:

```ruby
[% sponsors = [] %]  # sponsors list is now empty
```

## Combining Lists

Just like strings, there isn’t a limit to the number of lists you can create – you can use whatever is necessary to get the job done.

At some point, you may wish to merge two separate lists into one single list for further processing.

**merge**  Use `merge` to combine two or more lists together. You append the `merge` command to the first list, and send the other lists as arguments in the order you want them appended. For example:

```ruby
[% header = [
    'You are watching News 2 Wake Up',
    'Here is the latest news from News 2',
] %]

[% trailer = [
    'For more news, visit us online',
] %]

[% news = [
    'This is headline 1',
    'This is headline 2',
    'This is headline 3',
] %]

[% headlines = header.merge(news, trailer) %]
```

Now, `headlines` is a single list containing the follow strings in order:

- You are watching News 2 Wake Up
- Here is the latest news from News 2
- This is headline 1
- This is headline 2
- This is headline 3
- For more news, visit us online
Loop Processing
Virtually every LOGIC file will need to loop through data to display it. There are two main ways to loop: FOREACH and WHILE.

FOREACH and WHILE
As we illustrated earlier, the FOREACH loop iterates through items in a list, processing each one until its corresponding END has been reached.

```%
FOREACH game = Lottery [%
  \W\\Lotto[\% game.game_name %]
  \% game.results \%\CRLF
  \% SLEEP(3) %
\% END %]
```

Within the enclosing block, you can perform further tests based on the individual record or its position in the loop.

**size**  This can be tested on the full list to determine the number of items in the list. It’s useful if you want to show some lead-in animation or text but not if there is no data to display, as in:

```%
IF Sports.size > 0 [%
  \W\\news\Here are some scores...\CRLF
\% END %]
```

If you are inside of a loop, you can also use loop.size to generically refer to the current list. But since this test is usually performed before the loop starts, the name of the list can be used instead.

**loop.count**  When inside of a loop, this will return the number of times it has performed the loop, which means it starts at 1.

**loop.first**  This will evaluate true when used in an IF statement when the current loop iteration is the first.

**loop.last**  This will evaluate true when used in an IF statement when the current loop iteration is the last.

**loop.prev**  This will return the previous item in the list which is useful for performing A/B effects in VServer. (Expio is capable of remembering the previous data sent to a field for A/B effects.)

**loop.next**  This will return the next item in the list.

```%
IF loop.next.league != game.league [%
  The next game uses a different league
\% END %]
```
Note that returning the next item in the list doesn’t physically remove it from the list. So, it isn’t appropriate for advancing the list.

WHILE loops repeatedly process a block while a conditional expression evaluates true. The expression may be arbitrarily complex, but is generally used to make the same thing happen twice in the same LOGIC file, such as displaying a crawl twice:

```
[% x = 0 %]
[% WHILE x < 2 %]
  Show a crawl.
  [% x = x + 1 %]
[% END %]
```

LOGIC uses a failsafe counter to prevent runaway WHILE loops which would otherwise never terminate. If the loop exceeds 1,000 iterations, it will automatically terminate.

**Inter-Loop Flow Control**

**NEXT** The NEXT directive can be used in both FOREACH and WHILE loops to start the next iteration. When an END is reached, it automatically performs a NEXT, but there may be times when you wish to use some arbitrary logic to skip a record and thus wish to force the loop to proceed.

```
[% FOREACH game = Sports %]
  [% NEXT IF (game.league == 'NHL') %]
  Show all games but NHL.
  [% END %]
```

**LAST** The LAST directive works similarly to the NEXT directive, but instead of advancing the loop it exits it. BREAK can also be used as an alias for LAST. When debugging a LOGIC file, LAST is quite useful if you wish to only show a couple of sports scores (or headlines, or forecasts, etc.).

```
[% FOREACH game = Sports %]
  [% LAST IF (loop.count == 5) %]
  Show only the first four games
  [% END %]
```

**STOP** STOP can be used to tell NewsTicker to cease processing the current LOGIC file. It’s like a LAST for the entire file. It won’t stop the broadcast – it will just move NewsTicker on to the next module and LOGIC file in its list.

**Implied FOREACH using JOIN**

The join command performs an implicit FOREACH, returning the entire list with a specified delimiter between each element. An example of where this may be used is in crawl text.

```
\w1\news\n
[% headlines.join('<Image TV2_LOGO>') %]
```

```
\[ % CRLF %
```
The delimiter in this example is a crawl separator. If no delimiter is specified, then a space is used instead.

**Note** For `join` to work, the elements of the list must be scalars and not hashes. Hashes are standard for NewsTicker modules so you may need to perform pre-processing by creating a second list that just contains the strings you need, in order.

### Working with Parts of a List

The previous section discussed iterating over the entire list, which is a popular use of NewsTicker – you have a list of sports scores in the user interface and want to show all of them, so you perform a `FOREACH` to cycle through the whole list.

List elements can be accessed directly using the dot operator (`.`). You follow the list name with a dot and the element number you wish to use, starting at zero for the first element.

- `% sponsors.0 %` First element in the list `sponsors`
- `% sponsors.3 %` Fourth element in the list `sponsors`

It is also possible to access elements from the list using a variable that contains the index value you wish to use. Simply prefix the variable with a `$` character:

- `% index = 2 %`
- `% sponsors.$index %` Third element in the list `sponsors`

You can also work with a subset of the list by extracting it out.

**pop**

`LIST.pop`

This returns *and* removes the *last* item from the list, thus shortening the list by one.

- `% LIST = ['mcds.tga', 'bk.tga', 'wendys.tga'] %`
- `% LIST.pop %`

The `pop` command will write `wendys.tga` into the LOGIC file and make `LIST` now contain only `mcds.tga` and `bk.tga`. Therefore, another `pop` would return `bk.tga`.

**shift**

`LIST.shift`

Similar to `pop`, this returns *and* removes the *first* item from the list, thus shortening the list by one.

- `% LIST = ['mcds.tga', 'bk.tga', 'wendys.tga'] %`
The pop command will write mcds.tga into the LOGIC file and make bk.tga the new first item in the list.

splice

splice(LIST.splice(OFFSET, LENGTH, REPLACE))

Whereas pop and shift remove single items at the end or start of a list respectively, splice can selectively remove or replace part of a list.

The first argument defines the OFFSET in the list of the part to be removed, starting at 0 for the first item. With just that one argument, splice removes everything from that element onward, returning the removed items in a new list.

```
[% data = [
    'You are watching News 2 Wake Up',
    'Here is the latest news from News 2',
    'This is headline 1',
    'This is headline 2',
    'This is headline 3',
    'For more news, visit us online'
  ]
%
```

To return the three headlines and the trailer message:

```
[% headlines = data.splice(2) %]
```

You use 2 and not 3 since the first element is item 0, the second element is item 1, etc. The list data would now contain only the first two elements since splice removes the items it returns.

OFFSET can also be specified as a negative number, in which case it is counted backwards from the end of the list, with -1 being the last item in the list, -2 being the second to last item, etc.:

```
[% headlines = data.splice(-2) %]
```

headlines now contains:

```
This is headline 3
For more news, visit us online
```

If the LENGTH value is provided, splice will only remove the specified number of elements instead of the default, which is to remove everything from the OFFSET forward:

```
[% headlines = data.splice(2, 3) %]
```

That will just remove and return the three headlines, changing data to be:

```
You are watching News 2 Wake Up
Here is the latest news from News 2
For more news, visit us online
```
Finally, at the same time, you can replace the items you are removing with new elements. The number of new elements need not match the number you are splicing. REPLACE can either be the name of an existing list, or you can pass a list of items directly:

```
[% olddata = data.splice(1, 2, 'Breaking News', 'Man lands on the moon', 'Details to follow') %]
```

The data list now contains:
- You are watching News 2 Wake Up
- Breaking News
- Man lands on the moon
- Details to follow
- This is headline 3
- For more news, visit us online

Or, using an existing array:
```
[% newheader = [ 'News 2 is your safest place in bad weather', 'With exclusive 2-Trillion Doppler Radar', ] %]
```
```
[% oldheader = data.splice(0, 2, newheader) %]
```

data is now:
- News 2 is your safest place in bad weather
- With exclusive 2-Trillion Doppler Radar
- This is headline 1
- This is headline 2
- This is headline 3
- For more news, visit us online

**slice**

```
LIST.slice(FROM, TO)
```

The `slice` command works similar to the `splice` command with two exceptions: (1) it does not modify `LIST` since it returns the items without removing them, thus (2) you can’t REPLACE with the command.

It will return the items in the list between `FROM` and `TO`.

You can perform a `pop` or `shift` without removing the element using `slice`, as in:
```
[% elements = LIST.slice(-1, -1) %]
[% elements = LIST.slice(0, 0) %]
```

**unique**

```
LIST.unique
```

This method returns a copy of `LIST` with all duplicate values removed.
```
[% mylist = [ 1 2 2 4 3 3 4 5 6 6 1 ] %]
```
```
[% numbers = mylist.unique %]
```

The numbers list would contain 1, 2, 4, 3, 5, 6.
Re-Sorting a List

All module data is sorted in a meaningful manner when sent to the LOGIC file. In some cases, the order is fixed (such as headlines). In other cases, sort order is determined from within the configuration section of the user interface (such as sports) or, for version 4 modules, in a broadcast filter. But there are times when you may want to perform more complex sorting, or you may wish to use a different sort order in a particular runlevel without affecting the sort order of the same module in another runlevel. LOGIC allows you to re-sort from within the automation file.

**sort, nsort**

```
LIST.sort
LIST.sort('KEY')
LIST.nsort
LIST.nsort('KEY')
```

This method returns a copy of LIST in alphabetical (sort) or numerical (nsort) order, from lowest to highest. The following example illustrates the difference between the two:

```
[% list = ['0.1', '1', '02', '3', '010', '11',] %]

[% list = list.sort %]
Order is 0.1, 010, 02, 1, 11, 3

[% list = list.nsort %]
Order is 0.1, 1, 02, 3, 010, 11
```

When the items in the list are actually hashes (see next section, Lists vs. Hashes), an optional argument can be used to specify a sort key. Since virtually all NewsTicker data comes in this format (an array of hashes), this is particularly important.

```
[ % Finance = Finance.nsort('changePercent') %]
[ % FOREACH stock = Finance.reverse %]
    Stocks are returned from largest to least movers
[ % END %]
```

Note in the above example the use of reverse to swap the order. The nsort ordered the results lowest to highest. To display them highest to lowest, we tacked the reverse command on. The first two lines could also have been written as:

```
[ % Finance = Finance.nsort('changePercent').reverse %]
[ % FOREACH stock = Finance %]
```

Or, with more lines:

```
[ % Finance = Finance.nsort('changePercent') %]
[ % Finance = Finance.reverse %]
[ % FOREACH stock = Finance %]
```

It is possible to perform secondary sorting, but you have to be somewhat clever about it. Let’s say you want to group stocks first by
exchange, and then by symbol. (The Finance module lets you do this, but let’s pretend we wanted to do it the hard way.)

```
1 [% FOREACH stock = Finance %]
2   [% stock.sortvalue = "$stock.exchange$stock.symbol" %]
3 [% END %]
4 [% Finance = Finance.sort('sortvalue') %]
5 [% FOREACH stock = Finance %]
6   w\1\Stocks\[% stock.exchange %]
7   [% stock.symbol %]
8   [% stock.changeUpDown %][% stock.changeAmount %]
9   [% stock.last %]\[% CRLF %]
10 [% END %]
```

The sorting takes place in lines 1-4. In line 1, you instruct LOGIC to begin looping through each element. In line 2, instead of airing the data, you create a new scalar within the hash called sortvalue that contains the exchange and the symbol together in one field (NASDAQAAPL, NYSEDIS, NYSEGE, etc.) With this new field created, you can now sort on it in line 4, generating a newly reordered Finance list. Finally, the data is sent to the CG in lines 6-15.

Keep in mind you’ll never actually air the sortvalue field you created. It is merely there to sort on. The original exchange and symbol fields remain separated for use in your display.

**UNIQUEARRAYHASH**

UNIQUEARRAYHASH(ARRAY, 'KEY')

This method takes an ARRAY of hashes as its first argument, and the KEY within the hash you wish to find unique values for, and returns a new array of hashes with the KEY elements uniqued. The order of the returned array maintains the order of the original array.

Note that only the first occurrence of each hash element is returned. For example, the table below illustrates a sample Sports array:

<table>
<thead>
<tr>
<th>league</th>
<th>vt_name1</th>
<th>vt_score</th>
<th>ht_name1</th>
<th>ht_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NHL</td>
<td>Detroit</td>
<td>3</td>
<td>Phoenix</td>
</tr>
<tr>
<td>1</td>
<td>NHL</td>
<td>Chicago</td>
<td>2</td>
<td>St. Louis</td>
</tr>
<tr>
<td>2</td>
<td>MLB</td>
<td>NY Yankees</td>
<td>6</td>
<td>Minneapolis</td>
</tr>
<tr>
<td>3</td>
<td>NFL</td>
<td>Tampa Bay</td>
<td>20</td>
<td>Washington</td>
</tr>
<tr>
<td>4</td>
<td>NFL</td>
<td>Seattle</td>
<td>7</td>
<td>Cincinnati</td>
</tr>
<tr>
<td>5</td>
<td>NFL</td>
<td>Carolina</td>
<td>28</td>
<td>Tennessee</td>
</tr>
</tbody>
</table>
The next table illustrates the contents of Leagues based on the command
[% Leagues = UNIQUEARRAYHASH(Sports, 'league') %]:

<table>
<thead>
<tr>
<th>league</th>
<th>vt_name1</th>
<th>vt_score</th>
<th>ht_name1</th>
<th>ht_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>NHL</td>
<td>Detroit</td>
<td>3</td>
<td>Phoenix</td>
<td>1</td>
</tr>
<tr>
<td>MLB</td>
<td>NY Yankees</td>
<td>6</td>
<td>Minneapolis</td>
<td>1</td>
</tr>
<tr>
<td>NFL</td>
<td>Tampa Bay</td>
<td>20</td>
<td>Washington</td>
<td>14</td>
</tr>
</tbody>
</table>

From the original array, elements 0, 2 and 3 were returned and 1, 4 and 5 were skipped. In general, the command is only useful for examining one element of a hash (league in this example) and its common counterparts as the other elements will only be a subset of the larger data set.

**Lists vs. Hashes**

We’ve already defined a *scalar* as a variable that contains a single piece of information – either a string or number. And, we’ve said a *list* is a collection of two or more scalars in a prescribed order. In a list, also called an array, the name of the scalar is replaced with its numeric position in the list. So lists are useful when you want to group scalars and order is important.

*Hashes*, also called associative arrays, are like lists, except the names of each scalar are maintained. Since lists and hashes are two dimensional, you can’t have both a name and a position. Thus, hashes are useful when you want to group scalars and be able to pull out values based on the name of the contained scalar.

Whereas lists are initialized using the `[ ]` notation, hashes are defined using the `{ }` construct.

```
[% sponsors = {
    morning = 'chrysler.tga',
    noon = 'chevy.tga',
    5PM = 'ford.tga'
} %]
```

Within the `{ }` construct is each scalar, with the name (also referred to as *key*) to the left of the `=` (or `=>` if you prefer), and the *value* to the right.

Each value can be accessed directly using dot notation, as shown two ways below:

```
[% sponsors.morning %]
[% daypart = 'morning' %]
[% sponsors.$daypart %]
```
As is probably obvious, hashes cannot contain duplicate field names since the keys must be unique.

From a visual perspective, here is how data is stored in a list versus a hash:

\[
\begin{array}{|c|c|}
\hline
\text{Key} & \text{Value} \\
\hline
0 & chrysler.tga \\
1 & chevy.tga \\
2 & ford.tga \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Key} & \text{Value} \\
\hline
\text{morning} & chrysler.tga \\
\text{noon} & chevy.tga \\
\text{5PM} & ford.tga \\
\hline
\end{array}
\]

When writing LOGIC files, you work with both lists and hashes at the same time. Module data is technically returned as a list of hashes. Here is an example of the manner in which data is returned from NewsTicker for the Sports module. (We’ve removed several fields to make the example shorter and numbered each line for reference.)

```plaintext
[ % Sports = [
  { league = 'NFL',
    visit_team = 'Arizona',
    home_team = 'Kansas City',
    visit_score = '14',
    home_score = '24',
    period = 'F' },
  { league = 'NFL',
    visit_team = 'Minnesota',
    home_team = 'Baltimore',
    visit_score = '7',
    home_score = '3',
    period = '2' }
  ],
%]
```

You never see the data laid out like this in a LOGIC file since by the time you go to use the sports information, the lists and hashes have already been created. But, it is an example of how it would be laid out if manually written into the file.

Line 1 starts the Sports array. It’s an array because the games need to maintain a specific order. Because of this, you can perform a FOREACH on the Sports list. In this example, there are only two elements in the Sports list, and each element is itself a hash. They are a hash because (1) the
data needs to be grouped, and (2) it is easiest to pull the data out using a
label (name, key, etc.) instead of order (0, 1, 2).

Lines 2-7 are the first element in the list, a hash denoted using { ... }.
Lines 8-13 are the second element. In the LOGIC file, when you say
[% FOREACH game = Sports %], you are temporarily naming the currently
active hash game. At that point, dot notation is used to access the keys
within the hash (league, visit_team, period, etc.)

[% FOREACH game = Sports %]
  [% game.home_team %]
[% END %]

The above would print Kansas City and then Baltimore.

There are limited reasons to create your own hash inside of a LOGIC
file since, most of the time, order is important to maintain. There are
some added functions for hashes that we do not cover in this
documentation. Contact support if you have a specify inquiry on
further use of hashes.
Conditional Processing

Conditional processing is at the heart of making decisions based on real-time content and conditions.

**IF/UNLESS/ELSIF/ELSE**

The **IF** and **UNLESS** directives can be used to process or ignore a block of directives based on your own run-time condition.

```
[% IF game.desc_line1 %]
    Show comments about a sports game
[% END %]

[% UNLESS game.results %]
    Skip lottery results without any scores
[% END %]
```

Multiple conditions may be joined with **ELSIF** and/or **ELSE** blocks.

```
[% IF game.alt_period %]
    Show alternate status of game
[% ELSIF game.period %]
    Show status from drop-down box
[% ELSE %]
    Show scheduled time of the game
[% END %]
```

The following conditional and Boolean operators may be used:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>True if A is equal to B</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>True if A is not equal to B</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>True if A is less than B</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>True if A is not greater than B</td>
</tr>
</tbody>
</table>
> True if A is greater than B

>= True if A is not less than B

&& Use A if A is false, otherwise use B

and Same as &&

|| Use A if A is true, otherwise use B

or Same as ||

! True if A is not true

Note &&, || and ! are known as “short-circuit” operators because they allow you to make decisions based on multiple criteria without using nested IF statements. They skip (or short circuit) the evaluation of their right argument if they decide the left argument has already supplied enough information to decide the overall value. Instead of returning true or false, || and && actually return the last value evaluated. In the case of ||, that lets you select the first of a series of scalar values that happens to be true.

Conditions may be arbitrarily complex using precedence-based evaluation with parenthesis to explicitly determine evaluation order.

```plaintext
[% IF (game.visit_score > game.home_score) &&
  (game.final == 'Y') %]
  Change color of visiting team yellow

[% ELSIF (game.visit_score < game.home_score) &&
  (game.final == 'Y') %]
  Change color of home team yellow

[% ELSE %]
  Do nothing

[% END %]
```

Be careful that you always use == when evaluating if something is true in an IF/ELSFIF statement. If you accidentally use = when you mean ==, you’ll get unexpected behavior.

**SWITCH/CASE**

The SWITCH/CASE construct can be used to perform a multi-way conditional test. The SWITCH directive expects an expression which is first evaluated and then compared against each CASE statement in turn. Each CASE directive should contain a single value or a list of values which should match. CASE may also be left blank or written as [% CASE DEFAULT %] to specify a default match. Only one CASE matches – there is no drop-through between CASE statements.

```plaintext
[% SWITCH game.game_id %]
[ % CASE 'WIPowerBall' %]
  Show animation for PowerBall game
[ % CASE ['WIPick3', 'WIPick4'] %]
  Show Pick animation for Pick 3 or Pick 4 game
```
CASE 'WISuperCash' [%]
    Show animation for SuperCash game
[% CASE DEFAULT %]
    Show generic lottery animation
[% END %]

CASE works logically like an IF/ELSIF/ELSE statement, but is programmatically more efficient.
Flow Control

**Advancing the Loop**

There are two methods for arbitrarily moving around the runlevel loop: (1) by moving forward one module, or (2) by moving forward to the start of the next loop.

```plaintext
NEXTMODULE
NEXTMODULE()
```

This advances NewsTicker to the next module in the runlevel, exiting the current LOGIC file. The typical use of this command is in initialization files where, after unpausing the loop, you want NewsTicker to continue at the start of the next module versus somewhere in the middle of the previous one.

```plaintext
RESETLOOP
RESETLOOP()
```

This forces the current NewsTicker broadcast to move to the start of the next runlevel loop, or go back to the beginning of the current loop depending on how you look at it, and continue broadcasting. If you want to actually stop the broadcast, use STOPLOOP.

**Sleeping and Blocking**

After issuing commands to the character generator, you often need to wait a fixed amount of time before proceeding to allow time for viewers to read the content, or to wait until you receive notification from the CG that the crawl has left the screen or completed a certain operation.
**SLEEP**

**SLEEP(SECONDS)**

When performing cuts or “reads”, you typically send data to the CG and then do nothing for the length of time you want to display the data. **SLEEP** will wait the specified number of **SECONDS** before proceeding. Note that **SECONDS** can be an integer or a floating-point number.

In general, it is preferred to use **WAITFOR** commands (see below) that wait for a string of text that will never arrive, with the length of the timeout being the number of seconds you wish to sleep. This is because **WAITFOR** reads from the character generator’s send buffer which, if never read, may fill up and crash the CG’s automation protocol. Also, while sleeping, NewsTicker does not check to see if it has been paused or stopped whereas the **WAITFOR** command does.

**WAITFOR**

**WAITFOR(PATTERN, TIMEOUT)**

Used to wait for a specific response from the CG’s automation interface before proceeding, **WAITFOR** takes a regular expression as the first argument, and a timeout value for the second. NewsTicker will wait, monitoring the responses from the CG, until either the regular expression is satisfied, or the timeout value (in seconds) has elapsed.

For example:

```
[% WAITFOR('RTA\*', '30') %]
```

The above would wait 30 seconds for **RTA** from the CG. (Note that since it is passed directly as a regular expression, you need to escape the star. It also means you can do something like look for either **RTA** or **EOC**:)

```
[% WAITFOR('RTA\*|EOC\*', '30') %]
```

If you have used **SETTBUG** (see Chapter 6) to create a persistent time and temperature display, you must explicitly tell the **WAITFOR** command you wish to use it while waiting. You do this by appending **111TT** to the first argument. The above commands, rewritten displaying a time/temp bug, would be:

```
[% WAITFOR('RTA\*111TT', '30') %]
```

As described earlier, a **WAITFOR** can be used as a **SLEEP** command by setting the pattern to something that will never arrive, and using **TIMEOUT** as the length of time to sleep, as in:

```
[% WAITFOR('nothing', '4') %]
```

This is preferred since it will clear the CG’s response buffer on every call preventing a back-up of response messages that you may otherwise not care about, and can respond to a pause or stop command while waiting.
**Pausing and Stopping**

Once a runlevel has started broadcasting, it will continuously display information according to the LOGIC files unless a user (or automation command) instructs it otherwise.

Pausing a runlevel tells NewsTicker to temporarily stop what it is doing with the expectation that it will be resumed, such as during a commercial break or weather segment. Stopping a runlevel instructs NewsTicker to end the current runlevel and return to a state where a new runlevel can be started.

Pausing, resuming, and stopping are events that can be triggered and programmatically responded to from within the LOGIC files. This allows you to do things like restore a DVE to fullscreen, animate elements off-screen, or advance to the next module instead of resuming mid-module.

**Command Formatting**

The directives of what to do are typically set in the initialization file for the runlevel, but can be changed at any point from within any LOGIC file.

As you will see below, you will need to establish a **BLOCK** and **PROCESS** it into the respective commands in order to send multiple lines of instructions, for example:

```plaintext
[% BLOCK onstop %]
B\1\1[% CRLF %]
[% SLEEP(3) %]
D\1\1[% CRLF %]
[% END %]
[% onstopvalue = PROCESS onstop %]
[% nullval = SETONSTOP(onstopvalue) %]
```

The block is compiled when NewsTicker loads the LOGIC file in which it is contained. In the case of an initialization file, it will be compiled at the start of the runlevel. Normally, this would stop you from performing any **IF/THEN** statements since the statement would be performed once at compile-time (the start of the runlevel) instead of everytime the module is paused. To change that behavior, you can use `<% and %> from within the block, as in:

```plaintext
[% BLOCK onpause %]
<% IF DATESTRING('H') == '5' %>
  Do something related to the 5 AM hour
<% ELSIF DATESTRING('H') == '6' %>
  Do something related to the 6 AM hour
<% ELSE %>
  Do something else
<% END %]
[% END %]
```
If you had used [% %] instead of <%, then it would have used the hour value (DATESTRING('H')) from the time in which the LOGIC file was first launched versus the point at which the runlevel was actually paused.

**SETONPAUSE**  Sets commands to be executed when NewsTicker is paused.

**SETOFFPAUSE**  Sets commands to be executed when NewsTicker is resumed after a previous pause.

**SETONSTOP**  Sets commands to be executed when NewsTicker is running and is then instructed to stop.

**SETONPAUSESTOP**  Sets commands to be executed when NewsTicker is paused and then instructed to stop. This is separated from SETONSTOP because you are likely in a different state when you pause and then stop versus running and then stopping.

**Forcing State**  
In addition to manually pausing or stopping NewsTicker via the web interface or the intelliCommander, as well as automation control, you can force a change in state through the LOGIC files.

**PAUSELOOP**  
PAUSELOOP(SHOWONPAUSE, SHOWOFFPAUSE, RESUMETIME)  
This forces the current NewsTicker broadcast to pause. The command takes three optional arguments. SHOWONPAUSE (defaults to 1) is 1 if you want the commands usually executed during a SETONPAUSE to be displayed upon pausing, or 0 if you want them suppressed. SHOWOFFPAUSE (defaults to 1) is 1 if you want the commands usually executed during a SETOFFPAUSE to be displayed when the display resumes, or 0 if you want them suppressed. RESUMETIME is the number of seconds you want it to pause before resuming. If you don’t send this argument, NewsTicker will wait indefinitely for an operator to unpause it.

**STOPLOOP**  
STOPLOOP()  
This immediately stops the runlevel, executing any commands previously established in SETONSTOP. This is typically used to stop a school closing or severe weather display when there are no more active records or to terminate a “play once” runlevel like a snipe.
Date/Time/Temp Functions

NewsTicker has powerful built-in functions for integrating time and temperature with your displays. This not only allows you to eliminate a separate key from your broadcast chain, but opens possibilities for interaction between your bug and content display.

**CURRENTTEMP**

**CURRENTTEMP()**
This returns the current temperature of the city set in the Currents module as the default city. The value is numeric (without a degree symbol) for use in greater than/less than comparisons. In most fonts, you can display the degree symbol by appending \[ASCII(0176)\].

**CURRENTDATA**

**CURRENTDATA()**
This returns a reference to a hash containing all of the elements of the city set in the Currents module to be the default city. This gives you access to more than just the temperature.

```
[% curdat = CURRENTDATA %]
[% curdat.temperature %] (Same as [% CURRENTTEMP %])
[% curdat.windchill %]
```

**CLOCKTIME**

**CLOCKTIME()**
The time of day read from the server’s internal time clock, synched with the atomic clock. This time will reflect the time zone set up for your NewsTicker server, including daylight savings time if that has been selected.
This time cannot be used to make real-time comparisons in IF statements since it is inserted after all logic within the LOGIC file has been compiled. See COMPARETIME and DATESTRING for functions that allow for comparisons.

**SETTTBUG**

SETTTBUG(BUGCOMMANDS)

Used in conjunction with WAITFOR, NewsTicker can continue to display and update a time/temp bug while waiting for a response to a crawl. Without the use of this command, the time and temp wouldn’t be updated until after the crawl is complete. Typically, you would use the SETTTBUG command in the initialization LOGIC file:

```
[\% BLOCK timetemp \%]
\%tt\timetemp<\% CLOCKTIME \%>
<\% CURRENTTEMP %>
\% END \%
[\% timetempvalue = PROCESS timetemp %]
[\% nullval = SETTTBUG(timetempvalue) %]
```

You would then call for it within your individual LOGIC file WAITFOR commands by appending 111TT to the string you are waiting for. For example:

```
[\% WAITFOR('EOC 111TT', '30') %]
```

This is the same as [% WAITFOR('EOC ', '30') %] but displays whatever was assigned during the last SETTTBUG command.

**Advanced Functions**

NewsTicker supports additional data manipulations with date and time strings, including the ability to specify a format and compare two date stamps.

**DATESTRING**

DATESTRING FORMAT

This returns a date string according to the FORMAT specifications. This is useful for knowing what the current day of the week is, for instance, when performing IF/THEN commands based on that variable. NewsTicker only fetches this date and time at the beginning of each LOGIC file. So, if you were to ask for the current time, then sleep three seconds and ask again, it will be the same. But, the next time the LOGIC file loads, it will reset the date and time.

Here are the possible FORMAT options:

<table>
<thead>
<tr>
<th>Seq</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>A literal %</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>%a</td>
<td>Abbreviated weekday name (Sun, Sat)</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name (Sunday, Wednesday)</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name (Jan, Dec)</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name (January, April)</td>
</tr>
<tr>
<td>%c</td>
<td>Date and time as: Sat Nov 04 12:02:33 EST 1989</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month (01, 31)</td>
</tr>
<tr>
<td>%-d</td>
<td>Day of the month, no padding (1, 31)</td>
</tr>
<tr>
<td>%D</td>
<td>Date as mm/dd/yy (01/08/02) (See %x)</td>
</tr>
<tr>
<td>%e</td>
<td>Day of the month, blank padded ( 1, 31)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour, in military format (00, 08, 23)</td>
</tr>
<tr>
<td>%-H</td>
<td>Hour, in military format, no padding (0, 8, 23)</td>
</tr>
<tr>
<td>%I</td>
<td>Hour, in non-military format (01, 12)</td>
</tr>
<tr>
<td>%-I</td>
<td>Hour, in non-military format, no padding (1, 12)</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year (001, 366)</td>
</tr>
<tr>
<td>%-j</td>
<td>Day of the year, no padding (1, 366)</td>
</tr>
<tr>
<td>%k</td>
<td>Hour, in military format, blank padded ( 0, 8, 23)</td>
</tr>
<tr>
<td>%l</td>
<td>Hour, blank padded ( 1, 12)</td>
</tr>
<tr>
<td>%m</td>
<td>Month (01, 12)</td>
</tr>
<tr>
<td>%-m</td>
<td>Month, no padding (1, 12)</td>
</tr>
<tr>
<td>%M</td>
<td>Minute (00, 59)</td>
</tr>
<tr>
<td>%-M</td>
<td>Minute, no padding (0, 59)</td>
</tr>
<tr>
<td>%n</td>
<td>A newline</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM</td>
</tr>
<tr>
<td>%r</td>
<td>Time, in 12-hour format (01:39:59 PM)</td>
</tr>
<tr>
<td>%s</td>
<td>Seconds since 00:00:00, Jan 1, 1970, the epoch</td>
</tr>
<tr>
<td>%S</td>
<td>Seconds (00, 60)</td>
</tr>
<tr>
<td>%t</td>
<td>A horizontal tab</td>
</tr>
<tr>
<td>%T</td>
<td>Time, in 24-hour format (13:39:59)</td>
</tr>
<tr>
<td>%u</td>
<td>Week number of year with Sunday as first day of week (00, 53)</td>
</tr>
<tr>
<td>%-U</td>
<td>Week number of year with Sunday as first day of week, no padding</td>
</tr>
<tr>
<td>%v</td>
<td>Week number of year with Monday as first day of week (01 - 52)</td>
</tr>
<tr>
<td>%-V</td>
<td>Week number of year with Monday as first day of week, no padding (1 - 52)</td>
</tr>
<tr>
<td>%w</td>
<td>Day of the week with 0 representing Sunday (0, 6)</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of year with Monday as first day of week (00 - 53)</td>
</tr>
<tr>
<td>%-W</td>
<td>Week number of year with Monday as first day of week, no padding (0 - 53)</td>
</tr>
<tr>
<td>%x</td>
<td>Date represented as mm/dd/yyyy (01/08/2002) (See %D)</td>
</tr>
<tr>
<td>%y</td>
<td>Year, as two digits (00, 99)</td>
</tr>
<tr>
<td>%-y</td>
<td>Year, as two digits, no padding (0, 99)</td>
</tr>
<tr>
<td>%Y</td>
<td>Year, as four digits (1999, 2002)</td>
</tr>
</tbody>
</table>
You can combine the various elements above to create your own date format:

```
[% IF DATESTRING('%A') == 'Tuesday' %]
    Next play: Thursday
[% END %]

[% DATESTRING('Here are the headlines for %A, %B %-d ...') %]
```

The last example would show Here are the headlines for Friday, March 3 ...

**Note** This should not be used to display the current time in a bug as it only knows the time at the beginning of each module. It doesn’t update itself until you move to the next module in your loop.

**COMPARETIME**

`COMPARETIME(DATESTAMP)`

This function will take `DATESTAMP` and return how old it is (in seconds). `DATESTAMP` must be in the format of `YYYY-MM-DD HH:MM:SS`, which is the format that most of NewsTicker’s timestamps are in. If `DATESTAMP` is omitted, it returns the current time in epoch format (same as `DATESTRING('%s')`).

---

**%z** RFC-822 style numeric time zone (-0500, -0400)

**%Z** Time zone (EST, EDT, CDT)
Global Variables

Global variables are, by definition, those that can be seen in all LOGIC files. If you were to create a scalar in a LOGIC file, it would only hang around until the next LOGIC file is executed. The second file cannot normally reference scalars (or lists, or hashes) from the previous.

By creating it in the global namespace, your scalar (or list, or hash) will remain in memory for the duration of the runlevel. This is useful for maintaining the state of certain parts of your runlevel, such as the number of times the sponsor logo has been displayed.

There are two types of global variables: (1) those that are typed into NewsTicker’s Globals module, and (2) those that are created directly in the LOGIC files.

**NewsTicker Globals**

Content entered into NewsTicker’s Globals module is stored on disk and is retained over multiple runlevels. Each Global is stored in a numbered location in the module, and the value is recalled by using the assigned number.

\[
\text{GLOBAL} \quad \text{GLOBAL}_x() \\
\]

This returns the value from the Globals module where \(x\) relates to the appropriate global number. (i.e. \(\text{GLOBAL}_1()\), \(\text{GLOBAL}_3()\), etc.) Global variables are quite powerful. By using the web interface, newsroom personnel can easily update a data field that can always be seen while NewsTicker is broadcasting. This means that rotating sponsor logos is
as easy as telling a NewsTicker user to change GLOBAL2 from chevy.tga to ford.tga. You then simply use [% GLOBAL2 %] in your LOGIC file. Each time NewsTicker begins a new loop, it will check to see if this variable has changed, so you can easily rotate the sponsor manually.

Note that these values are read-only in the LOGIC file. You must use the NewsTicker interface to change the value.

**LOGIC Globals**
Outside of the NewsTicker user interface you can create global variables. These globals inside of LOGIC only exist for the duration of your runlevel.

To use LOGIC globals, simply prepend `global.` to the name of your scalar, such as:
```plaintext
[global.sponsor = 'ford.tga']
```

**Global Counter**
While you could create your own global counter by using something like `global.counter`, NewsTicker already includes one for lightweight applications.

**COUNTER**
`COUNTER()`
This first increments the internal counter, then returns the post-incremented counter total. `COUNTER` can cross multiple LOGIC files within the same runlevel and will continue to increment until you send a `CLEARCOUNTER` command.

```plaintext
[loopnum = COUNTER()] loopnum = 1
[loopnum = COUNTER()] loopnum = 2
[CLEARCOUNTER()] loopnum = 1
```

You can also perform an `IF` against it, which will increment and test at the same time:
```plaintext
[IF COUNTER == 5]
Do something
[CLEARCOUNTER]
[END]
```

**CLEARCOUNTER**
`CLEARCOUNTER()`
This resets the internal counter (referenced via the `COUNTER` command) to a value of 0.
Environment Information

Runlevel Information
When you start a runlevel, there is some environmental information available to you, allowing you to make real-time decisions based on current information. For example, you could use the same LOGIC file for multiple runlevels where only one minor change may be necessary, simply by checking what the current name is of the runlevel.

RUNLEVEL
RUNLEVEL()
This returns the name of the runlevel that the LOGIC file is currently associated with, which is obviously on air.

RLNUMBER
RLNUMBER()
Every runlevel has a unique ID that identifies it in the database. When you change the name of the runlevel, this ID remains the same. The only way for a runlevel to obtain a different number is to delete it and re-create it. For this reason, you may find it better to perform logical operations on RLNUMBER since it will survive a rename.

Note that this value is also visible when you are editing the runlevel’s loop through the NewsTicker GUI. It is listed as “Runlevel ID” on that screen.

RLMODULES
RLMODULES()
This returns an array of all modules in the runlevel, in the order they are to air. Thus, it corresponds to the values and order on the right side of the Edit Loop screen in the Broadcast module. In version 4 of the
Broadcast module, you can specify an Alias for each module in the Edit Loop screen. If present, the Alias will be used instead of the actual module name. Note that this command does not check to see if there is actual data to air in the module – just whether it is present in the runlevel list.

**RLMODULEINDEX**

```
RLMODULEINDEX()
```

Used in conjunction with RLMODULES, it returns the location within that array. Thus, if NewsTicker is currently airing the first module in RLMODULES, the RLMODULEINDEX value will be 0. If it is airing the third module, it would be 2.

**RL2XMODULES**

```
RL2XMODULES()
```

The most common use of RLMODULES is in presenting a tabbed display of what modules (or categories of information) are coming up next. But, if you are on the last module in the sequence (or second to last and are showing the next two or three modules forthcoming), you would have to write extra LOGIC code to look back at the start of the array.

Instead, you can use RL2XMODULES which is just like RLMODULES with an additional duplicated list of modules appended to it. That allows you to use a single array, and no fancy IF statements, to perform a tabbed display, even when it is at the end of the current runlevel loop.

---

**Output Device**

Decisions can also be made based on the output device that is currently airing the runlevel. Most of these relate to the appropriate settings from within the Configure screen of the Broadcast module.

**CGTYPE**

```
CGTYPE()
```

This is the type of output device, as designated from the popup box (e.g. File, Freedom, Infinit, etc.)

**CGBAUD**

```
CGBAUD()
```

Where applicable, this is the baud of the output device.

**CGPORT**

```
CGPORT()
```

This is the port specified in the configuration screen for the CG.

**CGNUM**

```
CGNUM()
```

This is the number assigned to the CG.

**ISCHILD**

`ISCHILD()`

This returns a Boolean value of `Y` or `N` depending on whether the broadcast was started from within another runlevel via the `SPAWNRUNLEVEL` command. (See *Advanced Functions.*)
The items discussed in this chapter involve more advanced programming concepts or are commands that may not be used by the typical user.

**Contact Closures**

The intelliCommander has four contact closure inputs, and four GPI outputs. The behavior of the inputs is determined in the Configure screen of the Broadcast module. The outputs can be fired from within a LOGIC file.

The `CONTACT` command takes three arguments. The first is the contact number or comma-separated numbers you wish to affect. The second argument is the action, which is either `OPEN`, `CLOSE`, or `MCLOSE` for momentary closure. The final argument is the intelliCommander ID you wish to affect, which defaults to 1.

For example, to close outputs two and three on intelliCommander one:

```
[ % CONTACT('2,3', 'CLOSE', '1') % ]
```

To perform a momentary closure of the first output on intelliCommander two:

```
[ % CONTACT('1', 'MCLOSE', '2') % ]
```

The intelliCommander will close a contact for a half-second upon a momentary closure. If you need a longer delay, you could do something like:

```
[ % CONTACT('4', 'CLOSE') % ]
[ % SLEEP('1.5') % ]
```
The above command would close contact four for one-and-a-half seconds on intelliCommander one.

**Custom intelliCommander Alarms**

In most installations, the intelliCommander is the hub of NewsTicker control and broadcast interaction. It can be configured to sound an alarm when the first school closes via automation, when severe weather hits the area, or an EAS message is received. Version 4 allows you to present your own message on the intelliCommander from within any LOGIC file.

The **ICMESSAGE** command takes three arguments. The first is the message to display on the intelliCommander, up to 22 characters. The second is the intelliCommander number on which to display the message. The third indicates whether to sound the aural alarm (1 = audible, 0 = silent).

As an example, let’s say you have a runlevel that displays a promotional snipe promoting your early news. The master control operator plays it at the specified time. You have included a test within the LOGIC file to not air the snipe if the content is older than four hours, which is certainly better than airing yesterday’s topicals. You wish to present a message to the master control operator letting him know the reason it didn’t air was because of old information. Your LOGIC may look something like this:

```%
[ % messages = [] %]
[ % FOREACH head = Headlines %]
  [ % NEXT IF compareTime(head.updated) > 14400 %]
  [ % messages.push(head) %]
[ % END %]
[ % IF messages.size == 0 %]
  [ % ICMESSAGE("Data too old", "1", "0") %]
  [ % STOPLOOP %]
[ % ELSE %]
  [ %# Air the snipe! %]
[ % END %]
```

On intelliCommander 1, the LCD will display the following without sounding an audible alarm:

![Message display](image)

Your message can be a maximum of 22 characters. The first 16 characters will appear on the first line, with the remaining characters on line two, after a hard break:
Match/Replace

LOGIC contains methods for finding certain occurrences of characters in a string, and acting upon it either by acknowledging whether it was found, returning a portion (or portions) of the matched data, or by replacing the matched text with something else. Conceptually, it is like the “Find” and “Replace” functions of word processors – except that it is monumentally more powerful.

The matching engine uses Perl regular expressions, which is a fancy way to say “pattern matcher.” Humans can detect many patterns very easily, but instructing a computer to do the same is another story. Unfortunately, regular expressions can be one of the toughest concepts for new programmers to grasp and use effectively, especially since Perl’s regular expression engine is one of the most advanced in existence. The `match` and `replace` functions within LOGIC do not expose all of the options a dedicated Perl program would, but it does give you the vast majority of what you will ever need.

There are many websites and entire books dedicated to the topic of writing regular expressions, also called a regex. We obviously won’t attempt to boil 450 pages down to four, but we will show you a few simple examples.

The most basic pattern would simply be a string of text, such as:

```%
% IF school.forced_status_name.match('Cancelled') %
   This school has cancelled something
% ELSE %
   This school appears not to have
% END %
%
```

The above example would look for the word “Cancelled” anywhere in the status name and, if it found it, would pass the test. This is a case of a verbatim match. You can loosen your search by making it case-insensitive:

```%
.match('(?i)Cancelled')
%
```

By putting `(?!)` at the start of your pattern, you have indicated a case-insensitive match, so “Cancelled”, “cancelled”, and “CaNcEllED” would all match.

There are two acceptable spellings for “cancelled” however, and with regular expressions, you can match either:

```%
.match('(?i)cancelled|canceled')
%
```
The pipe symbol (|) performs an “or”, as in “match if the string contains the case-insensitive word ‘cancelled’ or ‘canceled’.”

Here is a list of some of the more common commands used in creating regular expressions:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Force the expression to match at the start of the line</td>
</tr>
<tr>
<td>.</td>
<td>(period) Match any character</td>
</tr>
<tr>
<td>$</td>
<td>Force the expression to match at the end of the line</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>Escape character</td>
</tr>
<tr>
<td>( )</td>
<td>Group characters or patterns</td>
</tr>
<tr>
<td>[ ]</td>
<td>Define a character class</td>
</tr>
<tr>
<td>*</td>
<td>Match character or group 0 or more times (optional)</td>
</tr>
<tr>
<td>+</td>
<td>Match character or group 1 or more times (required)</td>
</tr>
<tr>
<td>?</td>
<td>Match 0 or 1 times (minimal match)</td>
</tr>
<tr>
<td>{n}</td>
<td>Match exactly n times</td>
</tr>
<tr>
<td>{n,}</td>
<td>Match at least n times</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Match at least n but no more than m times</td>
</tr>
</tbody>
</table>

Since the pattern is a string, you naturally need to use the escape character (\) to match one of the reserved characters. For example, if you wanted to match a period, you would need to write it as \. since a period without a backslash will match any character as described above. Same would be true of a ^ or $ or ( or, well, you get the point. There are some additional control sequences that have special meaning in regular expressions:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>The tab character</td>
</tr>
<tr>
<td>\r</td>
<td>The return character</td>
</tr>
<tr>
<td>\n</td>
<td>The newline character</td>
</tr>
<tr>
<td>\w</td>
<td>Any “word” character (alphanumeric plus “_”)</td>
</tr>
<tr>
<td>\W</td>
<td>Non-word character (the opposite of \w)</td>
</tr>
<tr>
<td>\s</td>
<td>White space (space, tab, or punctuation)</td>
</tr>
<tr>
<td>\S</td>
<td>Non-white space (the opposite of \s)</td>
</tr>
<tr>
<td>\d</td>
<td>Digit (0-9)</td>
</tr>
<tr>
<td>\D</td>
<td>Non-digit (the opposite of \d)</td>
</tr>
<tr>
<td>\b</td>
<td>Word boundary (basically “\w\s\w”)</td>
</tr>
<tr>
<td>\B</td>
<td>Non-(word boundary)</td>
</tr>
</tbody>
</table>

Going back to our earlier example, we could use a regular expression in the replace function to abbreviate a school closing display:

```% school.forced_status_name =
    school.forced_status_name.replace(  
        '(?i)cancelled|canceled', 'Cncld'
    )
%]```
The first argument is what it should match case-insensitively ("cancelled" or “canceled”), and the second argument is what it should replace any match with ("Cncld"). (Note that it is a global search and replace on the string.)

The replace command returns the modified string. Since the original is not modified, we simply made the original string equal to the result of the replace so that it would indeed change the original to be the version with the replaced text.

Another powerful feature of regular expressions and matching is the ability to return a subset of the string. You do this by grouping parts of the expression inside parentheses. Here is a highly contrived example:

```%
head = 'This is breaking news'
foo = head.match('(breaking|developing) news')
%
```

In this example, if the headline contained either the word “breaking” or “developing” just before the word “news”, it would match. The array `foo` contains the results of the match. It knows what part of the matched string `head` you want to extract since you put parentheses around it (highlighted above).

Note that even though there is only a single match, the values still go into an array. Thus, in this case, `foo.0` contains the word “breaking”.

Suppose we changed the expression to be:

```%
.match('[breaking|developing] (news)')
```

Then, `foo.0` would have contained “breaking” and `foo.1` would have been “news”.

Lottery games with bonus numbers are formatted with the regular numbers separated by hyphens, and the bonus number included in a set of parentheses, as in `1-2-3 (4)`. If you wanted to separate out the bonus number, you could use something like:

```%
IF (a = game.results.match('^([^\d]*) ([\d])'))
[results = a.0 %]
[bonus = a.1 %]
[ELSE %]
[results = game.results %]
[END %]
%
```

Here is another look at the regular expression:

```%
.match('
^([^\d]*)
\1\2 %)
```

Match all characters from the start of the string up to the first space (note the space is not illustrated here but is above)
By grouping parts of the regular expression with parentheses, we are able to control what is returned. So, the first grouping is the hyphenated numbers, and the second grouping is the bonus number without the surrounding parentheses. We could have changed the second part of the expression to read:

```
.match('^\1(\d*)')
```

In that case, it would have returned (4) instead of just 4.

Admittedly, we just scratched the surface of what regular expressions can do and anybody serious about using them should invest in a book or two on the topic. (O’Reilly publishes Mastering Regular Expressions as well as Regular Expression Pocket Reference.) Most of the reasons you would need to use a regular expression are mitigated by (1) available data elements within NewsTicker that have already been de-patternized, or (2) other functions such as split that can break up a string. However, the mere fact that you have this powerful tool at your disposal speaks to the infinite flexibility of LOGIC.

**Variable Interpolation**

Normally, you would access variables by use the normal [% %] syntax. But, LOGIC lets you access it directly by using plain text, allowing you to easily embed the value in a line of data. To do so, you use a simple $variable or ${variable} syntax.

For example, the following fragments have the same effect:

```
# using explicit directives
[% hline.display_line1 %] [% hline.display_line2 %]

# using interpolated variables
$hline.display_line1 $hline.display_line2
```

Generally, the [% %] notation is more readable since it clearly points out a variable.

In cases where the scope of the variable is ambiguous, you can (and should) use {} to scope it, such as:

```
${candidate.last_name}.tga
```

Or:

```
${sponsor.3}.tga
```
Without the curly braces, LOGIC would be looking for the nonexistent candidate.last_name.tga or sponsor.3.tga variable.

Where interpolation comes in really handy is when you have a complex data structure, and part of the structure is only known in a variable. For example, say you have a hash named leagues and it, in turn, contains a hash for each league with extra information, as in:

```%
leagues = {
    'NBA' = {
        name = 'NBA',
        logo = 'img_nba.tga' },
    'NCAAB' = {
        name = 'CBASK',
        logo = 'img_cbasketball.tga' },
}
%
```

Obviously, if you wanted the logo for NCAAB, you could use:

```
[ leagues.NCAAB.logo %]
```

But, you could also have the league in a variable of its own, and reference the value of that variable, as in:

```
[ league = 'NCAAB' %]
The icon is [ leagues.${league}.logo %]
```

LOGIC will know what you really want is the value of logo based on the value of league. If you were really sick, you could even do something like:

```
[ hash = 'leagues' %]
[ league = 'NBA' %]
[ key = 'logo' %]
The icon is [ ${hash}.${league}.${key} %]
```

**Sponsor Auditing**

It has already been illustrated that LOGIC files can rotate sponsor logos on just about any arbitrary criteria – that makes it quite powerful. That may be useless unless you have a way of tracking when the logo actually displayed.

NewsTicker’s SPONSORAUDIT command records when you sent the sponsor graphic to the CG, and which CG you sent it to. The information is then available from within the Sponsor Audit screen of the Broadcast module. It takes just one argument, which is typically the name of the sponsor (or the graphic filename if it is intuitive). Whatever you feed into it is what it will display in the report.

```
[ sponsor = 'ford.tga' %]
```

```w\_pg\Sponsor Template\[ sponsor %]\[ CRLF %]
```y\_pg[ CRLF %]```
Every time the sponsor is displayed, it will see the SPONSORAUDIT code and record fordf.tga.

If you have just a single sponsor for an entire broadcast, then you could simply put the statement into the initialization file so it only appears once for that broadcast, and not every time it gets displayed.

**Running Multiple Runlevels**

NewsTicker is capable of handling multiple runlevels simultaneously using the same common set of data (the central NewsTicker database). Obviously, since these are distinct runlevels, they can be associated with different looks. This allows you to be feeding multiple stations at once, or multiple streams of your own station, such as SD, HD and a digital sub-channel. Clearly, these could be started manually through the web GUI or via the intelliCommander.

It’s also possible to run more than one runlevel at a time on the same Freedom character generator, assuming you have taken care in creating the templates to ensure they are on independent layers. This is useful for complex layered displays, such as showing school closings in one runlevel while another is cycling through forecasts and traffic information.

LOGIC’s SPAWNRUNLEVEL command gives you the ability to launch a runlevel from within a runlevel, on either a separate CG or on the same one. The first argument is the runlevel number to start, which can be determined by reading the Runlevel ID value in the Edit Loop screen. The second argument is the CG number to start it on. The final argument involves whether the spawned runlevel should pause or stop with the current one, or whether it should remain independent. You do this by feeding it the CG number to observe. Note that the second and third arguments are optional – without them, the runlevel will be started on the current CG, pausing and stopping when the parent runlevel does.

There are four common scenarios in which the command is used:

**Two CGs, Independent Control** Runlevel 19 is started on CG 1, which starts runlevel 20 on CG 2. From this point forward, they act independent of each other. Pausing or stopping CG 1 has no effect on the runlevel airing on CG 2, and vice versa. In this case, everything starts together, but will be stopped separately. Runlevel 19 would issue the following command:

```
[% SPAWNRUNLEVEL('20', '2', '2') %]
```
Two CGs, Single Control  Runlevel 19 is started on CG 1, which starts runlevel 20 on CG 2. Pausing or stopping CG 1 will also pause or stop CG 2. In this case, everything starts together, and is controlled together, such as school closing displays on the same channel’s SD and HD feed. Runlevel 19 would issue the following command:

```
[% SPAWNRUNLEVEL('20', '2') %]
```

One CG, Independent Control  Runlevel 19 is started on CG 1. Runlevel 20 should be started on the same CG, but you want to maintain independent control over each runlevel. You must configure a new CG (e.g. CG 3) but use the exact same settings as CG 1. Pausing or stopping CG 1 will only affect runlevel 19 and pausing or stopping CG 3 will only affect runlevel 20. Runlevel 19 would issue the following command:

```
[% SPAWNRUNLEVEL('20', '3', '3') %]
```

One CG, Single Control  Runlevel 19 is started on CG 1, which starts runlevel 20 also on CG 1. Pausing or stopping CG 1 affects both runlevels simultaneously. In this case, everything starts together, and is controlled together, such as a multi-zoned school closing display. Runlevel 19 would issue the following command:

```
[% SPAWNRUNLEVEL('20') %]
```

Inside of a LOGIC file, the ISCHILD command returns a Boolean value of Y or N depending on whether the broadcast was started via SPAWNRUNLEVEL.

A parent runlevel can spawn several runlevels (multiple siblings), as can an already spawned runlevel (kids having kids, oh my).

Block and External Processing
The BLOCK ... END construct allows you to define LOGIC component blocks that can be processed as a unit. The most prevalent use of BLOCK is to define what should happen on a pause, resume, or stop in an initialization file. At the point the BLOCK is created, the code is not executed – in fact, it only runs when you issue the PROCESS command for that BLOCK. For example:

```plaintext
1 [% count = '0' %]
2 [% WHILE count < Finance.size %]
3 W\_stocks\Stocks\[
4 [% PROCESS nextrecord %]
5 [% PROCESS nextrecord %]
6 [% CLOCKTIME %]
7 [% CRLF %]
8 Y\_stocks\ [% CRLF %]
9 [% WAITFOR('nothing', '4') %]
10 [% count = count + 2 %]
```
In the above example, line 2 begins a \texttt{WHILE} loop that will test true until all of the stocks have been shown. Within the loop, it twice calls out to the \texttt{nextrecord} \texttt{BLOCK} via the \texttt{PROCESS} command (lines 4-5). Each time the request is made, the \texttt{BLOCK} shifts off the next stock to show, reducing \texttt{Finance} by one (line 15), and then outputs the automation command to display the data. As should be obvious, this display shows two stocks on-air at once.

The \texttt{PROCESS} command can also be used to execute some LOGIC code that is contained in an external file. This is most useful for things like time/temp bugs where you want to make a single change in one place versus in every LOGIC file from which it is called. For example:

\begin{verbatim}
[% PROCESS '/web/htdocs/formats/timetemp.fmt' %]
\end{verbatim}

The \texttt{INSERT} command works similar to \texttt{PROCESS} on an external file, except it makes no attempt to parse it. The contents, possibly including any embedded LOGIC directives, are inserted intact.

**Embedded Perl Processing**

LOGIC is a presentation language that provides a tremendous amount of flexibility within its built-in command set. However, there may be times when you want to go beyond what LOGIC offers you, and very occasionally, you may need the power of a full programming language within your LOGIC files.

The \texttt{PERL} directive allows you to embed a block of Perl code into a LOGIC file:

\begin{verbatim}
[% PERL %]
    print "Hello world\n";
[% END %]
\end{verbatim}

Note that embedding Perl should be considered a last resort. It is always more efficient and less messy to use a built-in command.

There are a number of special variables that are predefined within the \texttt{PERL} block, allowing access back into the calling LOGIC file. Most notably, the \texttt{$stash} variable contains a reference to the top-level data object, which is LOGIC's data engine where variables are managed.
Through this variable, values can be retrieved and updated, as in the example below where a Perl block checks to see if the character generator number defined in LOGIC’s cgnum variable is currently airing.

```
[% cgnum = '1' %]

[% Perl %]
    my $file = "/web/logs/ . $stash->get('cgnum');
    if (-e $file) {
        $stash->set(airing => 'yes');
    } else {
        $stash->set(airing => 'no');
    }

[% IF airing == 'yes' %]
    [% STOPLOOP %]
[% END %]

Exceptions may be thrown from within Perl blocks via die. If you don’t attempt to catch them, your LOGIC file will terminate prematurely. You can use a TRY block to prevent this from happening.

```

```
[% TRY %]
    [% PERL %]
    open(FILE, ">/tmp/sponsor") || die "Can't open file.";
    print FILE $stash->get('current_sponsor');
    close(FILE);

[% END %]

[% CATCH %]
    [% "Error: " _ error.info | stderr %]

[% END %]
```

In the previous example, if /tmp/sponsor couldn’t be opened for writing, the CATCH block would have received the error and, as instructed, sent “Error: Can’t open file” to the error log.

**Embedded LOGIC Options**

Version 4 of NewsTicker supports Embedded LOGIC Options. In addition to the standard module-based choices in the broadcast filters, you can add your own set of choices. This is a powerful feature letting end-users make changes using a GUI without having to manually edit declarations in LOGIC files.

Through these commands, you can present a free-flow text box, drop-down box, and multi-select scroll box within a filter. The choices picked by the end-user are then returned to the LOGIC file as a local variable (or global variable if you put it into the global namespace).

There are five commands in the syntax: FILTERYESNO, FILTERDROPDOWN, FILTERTEXTBOX, FILTERMULTITEXT and FILTERMULTISELECT. All five of these commands require you to pass a unique ID or key for each filter. These
IDs could be numbers or names that make sense to you, but they have to be unique in a given LOGIC file. (See Reserved IDs below.)

These functions also require you to provide a label. This label is what is presented to the end-user within the filter before making the selection. And, all three allow you to pass a default value in the event the individual doesn’t edit the filter.

You can present as many Embedded LOGIC Options to the end-user in a LOGIC file as you like. The order of the options is determined by the order in which you place the commands in your LOGIC file. You could spread the options out among several LOGIC files, or put them all into the initialization file. We recommend putting them at the top of the file outside of any blocks or loops.

**FILTERYESNO** This function displays a drop-down box to the user with two options: “Yes” or “No.” The values returned to the LOGIC file are either Y or N. The first argument is the unique ID for this element. The second argument is the textual description, or instructions, to appear in the Edit Filter screen. The third option is the default.

For example:

```
[% global.showtt = FILTERYESNO('usedve', 'Use DVE?', 'Y') %]
```

In the Edit Filter screen, under the Embedded LOGIC Options, the user will see the text “Use DVE?” with two options in a drop-down box: “Yes” or “No.” By default, “Yes” will be highlighted as Y is the third argument. When the LOGIC file is played in the runlevel, it will set global.usedve to either Y or N.

If you wish to present a Yes/No question to the end-user, this is the preferred method.

**FILTERDROPDOWN** This function works similar to FILTERYESNO except you get to specify the options presented. The first argument is the unique ID for this element. The second argument is the textual description, or instructions, to appear in the Edit Filter screen. The third option is the
default. The fourth argument is an array containing the value/label pairs for the drop box.

For example, the following is the same as \texttt{FILTERYESNO} as it presents “Yes” and “No” to the end-user, but returns either \texttt{Y} or \texttt{N} to the LOGIC file:

\[
\text{% global.showtt = FILTERDROPDOWN('tt', 'Show time and temp?', 'Y', ['Y', 'Yes', 'N', 'No']) %}
\]

Note that the array contains pairs. The first element (\texttt{Y}) is the value returned to the LOGIC file for option 1. The second element (\texttt{Yes}) is the label presented to the user in the filter for option 1. The third element (\texttt{N}) is the value returned to the LOGIC file when option 2 is selected. The fourth element (\texttt{No}) is the label presented to the user when selecting option 2. Obviously, that pattern continues.

You could also pass an array into the command, such as:

\[
\text{% options = ['sponsor1.tga', GLOBAL10, 'sponsor2.tga', GLOBAL11] %}
\]

\[
\text{% global.sponsor = FILTERDROPDOWN('sponsor', 'Choose your sponsor', options.0, options) %}
\]

In the previous example, the user would see whatever the value of \texttt{GLOBAL10} is for the first option and the value of \texttt{GLOBAL11} for the second option. The default is the first option (\texttt{options.0}).

\textbf{FILTERTEXTBOX}  This function provides a free-flow text box to the end-user. The first argument is the unique ID for this element. The second argument is the textual description, or instructions, to appear in the Edit Filter screen. The third argument is optional default text. The fourth option is the size of the box. And, the fifth argument is the maximum number of characters that can be entered into the box.

For example:

\[
\text{% global.sponsormsg = FILTERTEXTBOX('smsg', 'Sponsor Message:', 'Visit us online, anytime', '12', '75') %}
\]

When the filter is first opened, the default text \texttt{Visit us online, anytime} appears. The user can then delete that and type their own message, up to 75 characters.

\textbf{FILTERMULTITEXT}  This function provides a multi-line text box to the end-user. The first argument is the unique ID for this element. The second argument is the textual description, or instructions, to appear in the Edit Filter screen. The third argument is an array containing the
default line(s) of text. The fourth option specifies the number of lines of
text to present for entry (defaulted to 5). And, the fifth argument
specifies the width of the box (defaulted to 40).

For example:

```%
% global.sponsors = FILTERMULTITEXT('sponsors', 'Enter the
sponsor graphics to show:', ['nbclogo.tga', 'website.tga'], '8', '40')
%
```

The first line, in this case, will default to nbclogo.tga and the second line
will be website.tga. To specify no default, use the [ ] notation (empty
array) versus '' (empty string).

Note that global.sponsors, which is the variable used in the LOGIC file
to hold the lines the user has entered, is an array. So, if the user didn’t
set the filter, and instead the defaults are used, global.sponsors.0 would
be nbcbug.tga and global.sponsors.1 would be website.tga.

**FILTERMULTISELECT** This function presents a multi-selectable scroll box
to the end-user. The first argument is the unique ID for this element.
The second argument is the textual description, or instructions, to
appear in the Edit Filter screen. The third option is an array containing
the default(s). The fourth argument specifies how many elements
should be presented at one time. Scrollbars will be presented if the
number of elements exceeds the height. If you don’t specify a value, the
box will default to the number of elements in the box. The fifth option
is an array containing the value/label pairs for the scroll box.

For example:

```%
% global.sponsors = FILTERMULTISELECT('sponsors', 'Select which
sponsors to use in the rotation (hold CTRL to select more than one)',
[ 'nbcbug.tga', 'website.tga' ], '',
[ 'mcdonalds.tga', 'McDonalds',
  'ford.tga', 'Ford',
  'nbcbug.tga', 'Station Logo',
  'jimmys.tga', 'Jimmy Johns',
  'hdepot.tga', 'Home Depot',
  'website.tga', 'Website Address' ] )
%
```

Or, formatted for greater readability:

```%
% global.sponsors = FILTERMULTISELECT
  ('sponsors',
   'Select which sponsors to use in the rotation (hold CTRL
to select more than one)',
   ['nbcbug.tga', 'website.tga'],
   ['mcdonalds.tga', 'McDonalds',
    'ford.tga', 'Ford',
    'nbcbug.tga', 'Station Logo',
    'jimmys.tga', 'Jimmy Johns',
    'hdepot.tga', 'Home Depot',
    'website.tga', 'Website Address' ]
)%
```
The user will see the following options:

By default, it will rotate between “Station Logo” and “Website Address” as nbcbug.tga and website.tga are the defaults. To specify no default, use the [] notation (empty array) versus ’’ (empty string).

Just like the FILTERMULTITEXT command, an array is returned with the options selected by the user (or the defaults).

**Reserved IDs** The first argument passed to all of the above functions is the ID of that field. While these IDs must be unique within a single LOGIC file, they can be shared across multiple files. This is useful if your end-user switches from one LOGIC file to another. If you included the same Embedded LOGIC Option in both, such as the active sponsor, but used a different ID for each, then the end-user would have to re-select the active sponsor after switching LOGIC files. If, on the other hand, you kept the same ID in both LOGIC files for that same question, the value would be retained.

We recommend the following IDs be used for the purposes stated below. While you are free to choose your own, these are guidelines Newsroom Solutions follows internally when creating user LOGIC files, or working with files from the gallery.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cgdir</td>
<td>Working directory of the templates on the character generator</td>
</tr>
<tr>
<td>crawlspeed</td>
<td>Speed of a text crawl</td>
</tr>
<tr>
<td>crawllogo</td>
<td>Crawl separator command</td>
</tr>
<tr>
<td>crawloops</td>
<td>Number of loops to crawl before stopping</td>
</tr>
<tr>
<td>Template</td>
<td>Name of template</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>usett</td>
<td>Use time/temp in the display (Y/N)</td>
</tr>
<tr>
<td>usedve</td>
<td>Use DVE in the display (Y/N)</td>
</tr>
<tr>
<td>sponsors</td>
<td>Sponsor(s) to use in the display</td>
</tr>
</tbody>
</table>

More reserved IDs may appear in future documentation.
This chapter includes all of the module-specific data elements in NewsTicker. These data elements are available in any LOGIC file assuming the runlevel has aired a previous LOGIC file for the module you wish to use. (In most cases, data for a specific module is refreshed at the start of its LOGIC file.)

The data elements have been grouped by how common their use is. Fields that are internal or not commonly used are provided for use in logic operations within the file, or to display extra information (such as the PE ratio of a stock).

Unique identifiers are marked with a key notation and are useful when it is possible that the display name associated with the record could change, but you want to create logic operations for the record. The unique identifier will usually stay the same regardless of other changes in the record.

Currently, Newsroom Solutions is in the midst of provisioning a rolling upgrade from the 2-series of NewsTicker to the 4-series. This is made possible as a result of NewsTicker’s unique modular-based infrastructure. This provides for faster versioning since it allows older version 2 modules to co-exist with newer version 4 modules – you get new features as they are released, versus waiting for all modules to be converted. The version series for which the description applies is provided in the square notation after the module name below.

Version 4 modules provide for filtering when building a runlevel loop, allowing you to change how the information is presented to the LOGIC
file directly from the GUI. Those filters are described below, where appropriate.

Closings

The Closings LOGIC file is written different than the others since it requires real-time data. With other modules, such as Lottery, the games are loaded into memory and then the LOGIC file is compiled. Once the file is processed syntactically, it is handed over to the broadcast engine which then begins displaying the data. The engine is handed the instructions in one pass, which says something like “send the Pick 3 game and results to the CG, then sleep for four seconds, next send the Pick 4 game and results, sleep for another four seconds, and finally send the Keno game and results, and sleep a final four seconds.” At the time it is handed the instructions, it has all of the data it needs – it doesn’t look up the Keno results again just before it airs it.

There are several advantages to using this method: (1) it is more efficient programmatically, and (2) it allows you to perform more advanced logical operations on the data since it can be analyzed as a whole. For most NewsTicker modules, this is not a problem – grabbing the results for a half-dozen lottery games and then showing them is fine. The data doesn’t change frequently enough to warrant real-time lookups.

Closings is a different story. On a busy day, an entire loop may take 30 minutes to complete. If it took 20 minutes to go from A to S in the list, then “Smith School”, which closed 15 minutes ago, would not show until another loop. Clearly, that is not acceptable. LOGIC’s real-time engine for Closings prevents this from happening.

Performing Initial Sort

The most basic requirement of using the real-time engine is telling it the order in which you will be displaying the organizations. This could be as simple as alphabetical, or as complex as grouped by county and category. Because this order is established at the beginning of the file, and statuses will constantly update throughout the sequence, it is discouraged to sort based on a status. This shouldn’t pose much of a limitation since grouping by status is a disservice to your audience since they don’t have any clue when their organization will appear on your display.

Begin by telling NewsTicker the type of display you are using via the SETAIRTYPE command. Use a value of 1 if it will be a hot-change, such as cuts or dissolves, 2 if you will be crawling the information, 3 if it is a web
output, or 4 if the data is for an Always On display. Depending on this setting, the real-time engine will optimize itself for the display.

Next, sort the organizations using the sort method and feeding the results into SETAIRORDER. Below is a code snippet for a crawl display where the organizations are grouped by county, then arranged alphabetically:

```
[% SETAIRTYPE(2) %]

[% FOREACH record = Closings %]
  [% record.sortvalue = record.forced_county_name + record.forced_organization_name %]
[% END %]
[% resorted = Closings.sort('sortvalue') %]
[% nullval = SETAIRORDER(resorted) %]
```

### Setting Operational Blocks

There are three blocks that are necessary to be fed into SETSTARTLOOP, SETRECORDFOUND, and SETENDLOOP. As should be obvious, SETSTARTLOOP is the set of commands to run when the LOGIC file is first loaded — back to the lottery example, it would be everything before the FOREACH game = Lottery statement. Similarly, SETENDLOOP is what should be done once it has displayed all of the organizations — the code after the corresponding END statement in the lottery FOREACH example. Finally, SETRECORDFOUND is the code snippet to execute each time an active record is detected.

Your start and end blocks may look something like this:

```
[% BLOCK startloop %]
  \\C:/Templates/EarlyShow\\[% CRLF %
  \W\\Early Show\\HERE ARE THE LATEST CLOSINGS FROM CBS-2 ...
  \\\[% CRLF %
  \\\\\[% CRLF %
  [% END %]
[% startloopvalue = PROCESS startloop %]
[% nullval = SETSTARTLOOP(startloopvalue) %]

[% BLOCK endloop %]
  [% WAITFOR('EOC ','5000') %]
  [% END %]
[% endloopvalue = PROCESS endloop %]
[% nullval = SETENDLOOP(endloopvalue) %]
```

Your block for what to do when a record is found is based on the SETAIRTYPE variable described earlier. If you are hot-changing closings, then the real-time engine performs a look-up every time it advances the current pointer. Thus, if the next alphabetical school in the list closes while its alphabetical predecessor is being displayed, then it will correctly be the next record. However, if you are performing a crawl, the broadcast engine will perform a look-up, and then feed you a small set of active records (enough for approximately 30 seconds). This
optimization assures you have the latest information without over-taxing the server.

Because a hot-change display produces a single record and a crawl produces multiple records ordered in an array, the behavior of this block differs slightly based on your selection.

In a hot-change display, the special variable 
CREC  (for “current record”) is populated with the currently active record:

```plaintext
[% BLOCK recordfound %]
\W\clo\Closings\n  <\% CREC.forced_organization_name %>
  <\% CREC.forced_status_name %>
  <\% CREC.status2_name1 %>[\% SPACE %]
  <\% CREC.comments_line1 %>\[\% CRLF %]
\W\clo\[\% CRLF %]

  <\% WAITFOR('sleeping','4') %>
  [\% END %]
  [\% recordfoundvalue = PROCESS recordfound %]
  [\% nullval = SETRECORDFOUND(recordfoundvalue) %]
```

**Note** Within the blocks itself, it is important to put those commands that must be processed in real-time, such as the actual record data and any logic you may perform on it, into the `<% %>` syntax vs. `[ % ]`. Otherwise, those commands would be executed when you performed the PROCESS on the block instead of waiting until the block is actually used on a record.

In a crawl display, the special array CRECS  (for “current records”) is populated with the currently active records:

```plaintext
[% BLOCK recordfound %]
\W\2\Early Show\n  <\% FOREACH record = CRECS %>
    <\% record.forced_organization_name %>:<\% record.status_name1 %>
  [\% SPACE %]<\% record.comments_line1 %>\[\% CRLF %]
\W\2\[\% CRLF %]
  <\% WAITFOR('RTA|EOC','5000') %>
  [\% END %]
  [\% recordfoundvalue = PROCESS recordfound %]
  [\% nullval = SETRECORDFOUND(recordfoundvalue) %]
```

The idea is that you’ll be feeding it small chunks of crawl data at a time, waiting until the last possible second to feed it more so that it is as timely as possible – this, instead of sending the entire crawl text at one time.
**Using Just In**

The Just In feature is typically configured within the runlevel’s initialization file. There are two primary things NewsTicker must know in order to perform the Just In display: (1) what code should it execute, and (2) what queue should it use.

Similar to the blocks described in the previous section, the special array JustIn will contain an ordered list of all records in the queue to be displayed. You create a block and feed the contents into SETJUSTINLOOP, keeping in mind the need for the <% %> syntax:

```%
% BLOCK justin %
\%\_clo2\%JI\Just In...[\% SPACE \%][\% SPACE \%]

<% FOREACH record = JustIn %>
  <% record.forced_organization_name %>: [\% SPACE %]<% record.status_name1 %>
  <image 15 -2 15 Logo2>
<% END %>
\% CRLF \%
\% Clo2\% CRLF \%
<% WAITFOR('EOC Display2 "+",'5000') %>

\% Clo2\% CRLF \%

<% END %>
[% justinloopvalue = PROCESS justin %]
[% nullval = SETJUSTINLOOP(justinloopvalue) %]
```

While the above example is a crawl, it could easily be a hot-change display as well. If you do display two crawls at a time, you may wish to slightly speed-up the one used for Just In since it tends to read easier when two crawls aren’t at the same speed.

Once you’ve passed the code into SETJUSTINLOOP, we need to tell NewsTicker what queue it should be reading from:

```%
% USEJUSTIN('1') %
```

If you are just broadcasting to a single CG, then you will likely use queue 1. But, for cases where you are showing closings on multiple, independent CGs, each needs to use its own queue so that showing a Just In record on one doesn’t prevent it from showing on another.

**Data Elements**

There are a couple levels of entering a status that must be understood in the context of the data elements for Closings. The first level involves defined versus free-flow.

**Defined Status** Defined statuses are those chosen by entering a corresponding number (telephony interface) or popup box (web interface). These defined statuses may prompt for further information, such as the length of a delay or the time the organization will be
opening. After choosing a primary defined status, a caller may optionally choose a secondary defined status, if you have configured it this way. Note that STATUS\_NAME2 is not a continuation of STATUS\_NAME1—when used, it is an alternative formatting of the status name. (The same is true for STATUS2\_NAME1 and STATUS2\_NAME2.)

**Free-Flow Status**  Free-flow statuses are those entered manually by the organization and can only be entered via the web (or by the station using the NewsTicker GUI). This allows an organization to provide more detailed information that is specific to their audience, when a defined status will not work.

An organization can use either a defined status or a free-flow status, but not both at the same time. The free-flow status is an *alternative* to the defined status. This manual information will be populated in the ALT\_STATUS\_TEXT element and the STATUS\_NAMEx elements will be empty.

**Comments**  Additional free-flow information can be entered into the comments field to provide more information about the active status. Comments can be applied to either a defined or free-flow status.

Thus, it is possible to have one line of information (just a primary defined status or free-flow status with no comments), two lines of information (a primary and secondary defined status with no comments, a primary defined status with comments, or a free-flow status with comments), or three lines of information (a primary and secondary defined status with comments).

Comments are passed in the COMMENTS\_LINE1 element.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>forced_organization_name</td>
</tr>
<tr>
<td>forced_status_name</td>
</tr>
<tr>
<td>status2_name1</td>
</tr>
<tr>
<td>comments_line1</td>
</tr>
<tr>
<td>forced_county_name</td>
</tr>
<tr>
<td>forced_category_name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec_id (Key)</td>
</tr>
<tr>
<td><strong>organization name1</strong></td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td><strong>organization name2</strong></td>
</tr>
<tr>
<td><strong>organization name3</strong></td>
</tr>
<tr>
<td><strong>alt_organization_name</strong></td>
</tr>
<tr>
<td><strong>address1</strong></td>
</tr>
<tr>
<td><strong>address2</strong></td>
</tr>
<tr>
<td><strong>city</strong></td>
</tr>
<tr>
<td><strong>state</strong></td>
</tr>
<tr>
<td><strong>zipcode</strong></td>
</tr>
<tr>
<td><strong>contact_name</strong></td>
</tr>
<tr>
<td><strong>contact_tx</strong></td>
</tr>
<tr>
<td><strong>contact_email</strong></td>
</tr>
<tr>
<td><strong>organization homepage</strong></td>
</tr>
<tr>
<td><strong>notes</strong></td>
</tr>
<tr>
<td><strong>status_code</strong></td>
</tr>
<tr>
<td><strong>status_dow</strong></td>
</tr>
<tr>
<td><strong>userdef1</strong></td>
</tr>
<tr>
<td><strong>status_name1</strong></td>
</tr>
<tr>
<td><strong>status_name2</strong></td>
</tr>
<tr>
<td><strong>status_line3</strong></td>
</tr>
<tr>
<td><strong>alt_status_text</strong></td>
</tr>
<tr>
<td>Feature</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>status2_code</td>
</tr>
<tr>
<td>status2_name1</td>
</tr>
<tr>
<td>status2_name2</td>
</tr>
<tr>
<td>status2_name3</td>
</tr>
<tr>
<td>comments_line2</td>
</tr>
<tr>
<td>comments_line3</td>
</tr>
<tr>
<td>county_name1</td>
</tr>
<tr>
<td>county_name2</td>
</tr>
<tr>
<td>alt_county_name</td>
</tr>
<tr>
<td>cat_code</td>
</tr>
<tr>
<td>category_name1</td>
</tr>
<tr>
<td>category_name2</td>
</tr>
<tr>
<td>updated</td>
</tr>
<tr>
<td>expiration</td>
</tr>
</tbody>
</table>

The expiration time is provided for those organizations that have yet to expire. Once the expiration time has been reached, the record will be purged automatically so you don’t need to test it within your LOGIC file.

**Currents**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_name1</td>
<td>City name as entered by user</td>
<td>Newark, Flint</td>
</tr>
<tr>
<td>condition_name1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**COMMONLY USED ELEMENTS**
| **Current condition as entered by user** | Partly cloudy, Rain |
| **cond_pic** | Icon keystroke as configured by user | C, R |
| **temperature** | Current temperature | 54, -9 |
| **wind** | Wind direction | W, NE |
| **wind_speed** | Wind speed | 8, 2 |
| **windchill** | Wind chill temperature | -2, 0, 12 |
| **hindex** | Heat index temperature | 62, 81, 109 |
| **sunrise** | Sunrise time for city, in local timezone | 6:32 |
| **sunset** | Sunset time for city, in local timezone | 7:12 |

**OTHER ELEMENTS**

| **station (Key)** | NWS reporting station ID code | KEWR, KFNT |
| **rcity** | Reporting city, as fed from NWS | Newark |
| **rstate** | Reporting state, as fed from NWS | NJ |
| **condition** | Unformatted condition as fed by TickerWire | PTCLDY, TSTRM |
| **dewpoint** | Current dewpoint | 26, 19 |
| **pressure** | Barometric pressure | 30.04, 26.00 |
| **status** | Order of the city in the display | 1, 2, 3 |
| **updated** | Date/time record was last updated or observation was made by the NWS | 2004-03-12 00:05:10 |

**Elections**

Not unlike Closings, Elections works slightly different than the rest. Each element in the Elections list is a race, suitably processed with a FOREACH command. But, within each iteration is a second list (candidates) which contains each candidate. That second list needs to be iterated over as well:
When working with full screen LOGIC files, only one race is sent at a time. You can either iterate over a single element in the `Elections` list, or you can simply pull the race out directly, as in:

```
[% race = Elections.0 %]
```

### Elections

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>race_name1</code></td>
<td>User formatted name of race, line 1</td>
</tr>
<tr>
<td><code>race_name2</code></td>
<td>User formatted name of race, line 2</td>
</tr>
<tr>
<td><code>race_name3</code></td>
<td>User formatted name of race, line 3</td>
</tr>
<tr>
<td><code>race_name4</code></td>
<td>User formatted name of race, line 4</td>
</tr>
<tr>
<td><code>jurisdiction_name1</code></td>
<td>Name of assigned jurisdiction, line 1</td>
</tr>
<tr>
<td><code>jurisdiction_name2</code></td>
<td>Name of assigned jurisdiction, line 2</td>
</tr>
<tr>
<td><code>num_seats</code></td>
<td>Number of seats (potential winners)</td>
</tr>
<tr>
<td><code>precincts_reporting</code></td>
<td>Number of precincts reporting</td>
</tr>
<tr>
<td><code>precincts_total</code></td>
<td>Total number of precincts</td>
</tr>
<tr>
<td><code>pct_precincts_reporting</code></td>
<td>Percentage of precincts reporting</td>
</tr>
<tr>
<td><code>race_id</code></td>
<td>Unique ID of race</td>
</tr>
<tr>
<td><code>jurisdiction_id</code></td>
<td>Unique ID of assigned jurisdiction</td>
</tr>
<tr>
<td><code>fs_format</code></td>
<td>User-assigned full-screen LOGIC file</td>
</tr>
<tr>
<td><code>prompt_lines1</code></td>
<td>Anchor prompt lines, line 1 – can be used with tPrompt or exported to web site for more in-depth information</td>
</tr>
</tbody>
</table>

**COMMONLY USED ELEMENTS**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>race_name1</code></td>
<td>Smith City Mayor-R</td>
</tr>
<tr>
<td><code>race_name2</code></td>
<td>Smith City</td>
</tr>
<tr>
<td><code>race_name3</code></td>
<td>Mayor</td>
</tr>
<tr>
<td><code>race_name4</code></td>
<td>Republican</td>
</tr>
<tr>
<td><code>jurisdiction_name1</code></td>
<td>Franklin County</td>
</tr>
<tr>
<td><code>jurisdiction_name2</code></td>
<td>Smith City</td>
</tr>
<tr>
<td><code>num_seats</code></td>
<td>1, 2</td>
</tr>
<tr>
<td><code>precincts_reporting</code></td>
<td>36, 50</td>
</tr>
<tr>
<td><code>precincts_total</code></td>
<td>40, 50</td>
</tr>
<tr>
<td><code>pct_precincts_reporting</code></td>
<td>0, 88, 100</td>
</tr>
<tr>
<td><code>race_id</code></td>
<td>100, 5005</td>
</tr>
<tr>
<td><code>jurisdiction_id</code></td>
<td>1, 2</td>
</tr>
<tr>
<td><code>fs_format</code></td>
<td>2cand.fmt</td>
</tr>
<tr>
<td><code>prompt_lines1</code></td>
<td>This is Jones' third run for mayor.</td>
</tr>
<tr>
<td><code>prompt_lines2</code></td>
<td></td>
</tr>
</tbody>
</table>
Ortiz is running for a second term.

### ranked
Manner in which candidates are ordered (V=Vote total; O=Order listed; L=Last name; F=First name)

### allcand_votes
Total number of votes cast

### updated
Date/time record was last updated

### candidates

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first_name</strong></td>
</tr>
<tr>
<td><strong>last_name</strong></td>
</tr>
<tr>
<td><strong>incumbent</strong></td>
</tr>
<tr>
<td><strong>winner</strong></td>
</tr>
<tr>
<td><strong>party_abbr</strong></td>
</tr>
<tr>
<td><strong>party_name1</strong></td>
</tr>
<tr>
<td><strong>party_name1</strong></td>
</tr>
<tr>
<td><strong>vote_total</strong></td>
</tr>
<tr>
<td><strong>pct_total</strong></td>
</tr>
<tr>
<td><strong>pic_id</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>candidate_id</strong></td>
</tr>
</tbody>
</table>
**Finance**

All stocks and indices are returned in the Finance list. You may use `dataType` to differentiate between the various types and filter out those you don’t wish to show.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>symbol</strong></td>
<td>(Key)</td>
</tr>
<tr>
<td>Symbol of security</td>
<td>GM, AAPL, .DJC</td>
</tr>
<tr>
<td><strong>dataType</strong></td>
<td></td>
</tr>
<tr>
<td>Type of data set (s: stock, i: index)</td>
<td>s, i</td>
</tr>
<tr>
<td><strong>name1</strong></td>
<td></td>
</tr>
<tr>
<td>User-editable name of security</td>
<td>General Motors, Dow Jones Composite</td>
</tr>
<tr>
<td><strong>last</strong></td>
<td></td>
</tr>
<tr>
<td>Last trade value</td>
<td>16.69, 30.75</td>
</tr>
<tr>
<td><strong>changeAmount</strong></td>
<td></td>
</tr>
<tr>
<td>Amount of change since previous close</td>
<td>0.19, 3.12</td>
</tr>
<tr>
<td><strong>changePercent</strong></td>
<td></td>
</tr>
<tr>
<td>Percentage change since last close</td>
<td>1.15, 0.12</td>
</tr>
<tr>
<td><strong>changeUpDown</strong></td>
<td></td>
</tr>
<tr>
<td>A plus, minus, or nothing depending on what should precede changeAmount and changePercent</td>
<td>+, -, [NULL]</td>
</tr>
<tr>
<td><strong>ranking</strong></td>
<td></td>
</tr>
<tr>
<td><em>Not currently implemented</em></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>exchange</strong></td>
<td></td>
</tr>
<tr>
<td>The abbreviation of the exchange</td>
<td>AMEX, NASDQ, NYSE</td>
</tr>
<tr>
<td><strong>lastTradeTime</strong></td>
<td></td>
</tr>
<tr>
<td>Last time stock was traded</td>
<td>2001-03-08 14:02:00</td>
</tr>
<tr>
<td><strong>pe</strong></td>
<td></td>
</tr>
<tr>
<td>PE ratio</td>
<td>3</td>
</tr>
<tr>
<td><strong>volume</strong></td>
<td></td>
</tr>
<tr>
<td>Volume of trades for the day</td>
<td>487600</td>
</tr>
<tr>
<td><strong>previousClose</strong></td>
<td></td>
</tr>
<tr>
<td>Previous day’s closing value</td>
<td>15.32</td>
</tr>
<tr>
<td><strong>todayOpen</strong></td>
<td></td>
</tr>
<tr>
<td>Today’s open value</td>
<td>13.25</td>
</tr>
<tr>
<td><strong>todayHigh</strong></td>
<td></td>
</tr>
<tr>
<td>Today’s highest value</td>
<td>18.65</td>
</tr>
<tr>
<td><strong>todayLow</strong></td>
<td></td>
</tr>
<tr>
<td>Today’s lowest value</td>
<td>12.13</td>
</tr>
<tr>
<td><strong>todayTrades</strong></td>
<td></td>
</tr>
<tr>
<td>Number of trades for the day</td>
<td>2165</td>
</tr>
<tr>
<td><strong>tmonHigh</strong></td>
<td></td>
</tr>
<tr>
<td>52-week high</td>
<td>24.50</td>
</tr>
</tbody>
</table>
Forecasts

Forecast data is returned starting with the first city’s daypart. If you have three dayparts configured, the data is returned in the following order:

City 1, Daypart 1
City 1, Daypart 2
City 1, Daypart 3
City 2, Daypart 1
City 2, Daypart 2
City 2, Daypart 3

A popular option is to put all dayparts you need in the module, and then leave the forecast description blank for those you don’t want to show currently. Thus, in your LOGIC file, you would skip dayparts that contain no forecast data.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>city_label</td>
</tr>
<tr>
<td>forecast_label</td>
</tr>
<tr>
<td>forecast</td>
</tr>
<tr>
<td>pic</td>
</tr>
<tr>
<td>temp_label1</td>
</tr>
<tr>
<td>temp1</td>
</tr>
<tr>
<td>temp_label2</td>
</tr>
<tr>
<td>temp2</td>
</tr>
<tr>
<td>temp_label3</td>
</tr>
<tr>
<td>temp3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>pic_label</td>
</tr>
</tbody>
</table>
Description chosen in pop-up menu | Rain, Snow
---|---
updated | Date/time record was last updated | 2001-02-15 16:03:12

### Headlines

By default, headline data is returned in the order it appears in the GUI. See *Filter Options* below for information on how to select categories and a new order.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_line1</td>
</tr>
<tr>
<td>display_line2</td>
</tr>
<tr>
<td>display_line3</td>
</tr>
<tr>
<td>category_line1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>head_id</td>
</tr>
<tr>
<td>hcat</td>
</tr>
<tr>
<td>display_lines</td>
</tr>
<tr>
<td>updated</td>
</tr>
</tbody>
</table>

### Filter Options

The categories sent to the LOGIC file, and their ordering, is determined by the filter attached to that instance of the module within the runlevel. By default, all categories are sent in the order in which they appear in the GUI.

**Data Categories**  This option allows you to determine whether you want to show all categories, just a select grouping, or all categories except for specific ones. By choosing to show all categories except for ones you designate, new categories will automatically appear in your display as you add them.

**Category Ordering**  By default, headline categories are sent to the LOGIC file in the order they appear in the GUI, which you can change by clicking *Configure* in the Headlines module. You can override that by
selecting Use custom sort order and arranging the categories in your desired order.

**Lottery**

All lottery games are returned, regardless of when the numbers were drawn. You may wish to skip over older games by applying logic to the updated field.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>game_id (Key)</td>
</tr>
<tr>
<td>Unique ID of lottery game</td>
</tr>
<tr>
<td>game_name</td>
</tr>
<tr>
<td>User entered name of game</td>
</tr>
<tr>
<td>results</td>
</tr>
<tr>
<td>Results of game</td>
</tr>
<tr>
<td>updated</td>
</tr>
<tr>
<td>Date/time record was last updated</td>
</tr>
</tbody>
</table>

SevereWx

SevereWx LOGIC files are written to display either a bug or crawl. In general terms, a “bug” may contain a map, radar, and/or text cycling through the active conditions (e.g. Tornado Watch/Jefferson Co./Until 5:00 pm; Flash Flood Warning/Montgomery Co./Until 5:30 pm). A “crawl” contains sentences describing the conditions and what to do, sometimes including the full National Weather Service bulletin.

Within the filter, you specify whether the display being executed in the LOGIC file is optimized for a bug or crawl. Based on that choice, different data elements are returned to the LOGIC file.

**Data Elements**

The data elements are listed below. Those marked Crawl Only are available when the filter is set to Optimized for Crawl. Elements marked Bug Only are available when set to Optimized for Bug. All others are available regardless of optimization.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition (Key)</td>
</tr>
<tr>
<td>Unique condition ID</td>
</tr>
<tr>
<td>forced_condition_name</td>
</tr>
<tr>
<td>Condition name specified by user/filter</td>
</tr>
</tbody>
</table>

| game_id (Key)                                              |
| Unique ID of lottery game                                   |
| MIPick3Eve, IAKeno                                         |
| game_name                                                  |
| User entered name of game                                  |
| Daily (MI), Keno                                           |
| results                                                    |
| Results of game                                            |
| 2-5-9, 1-5-23-8 (1)                                        |
| updated                                                    |
| Date/time record was last updated                           |
| 2001-02-15 16:03:12                                        |

| condition (Key)                                             |
| Unique condition ID                                         |
| TOW, SVA, FLY, O1W                                         |
| forced_condition_name                                       |
| Condition name specified by user/filter                     |
| Severe Thunderstorm                                        |
### Crawl Text (Crawl Only)
- An array containing the elements text (the full crawl text) and data_type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>crawl_text</td>
<td>Warning</td>
</tr>
<tr>
<td>text</td>
<td>The NWS has issued a...</td>
</tr>
</tbody>
</table>

### Forced Zone Name (Bug Only)
- Zone name; NWS default if not assigned

<table>
<thead>
<tr>
<th>Zone Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>forced_zone_name</td>
<td>Westmoreland</td>
</tr>
</tbody>
</table>

### Forced Expire Time (Bug Only)
- Formatted expiration time

<table>
<thead>
<tr>
<th>Expiration Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>forced_expire_time</td>
<td>9:00 pm Tomorrow, 8:53 am</td>
</tr>
</tbody>
</table>

### Texture Translate (Bug Only)
- Calculated Expio texture translation values for radar zooming (string)

<table>
<thead>
<tr>
<th>Texture Translate</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>texture_translate</td>
<td>.177 .161 1</td>
</tr>
</tbody>
</table>

### Texture Scale (Bug Only)
- Calculated Expio texture scale values for radar zooming (string)

<table>
<thead>
<tr>
<th>Texture Scale</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>texture_scale</td>
<td>-.288 .276 0</td>
</tr>
</tbody>
</table>

### Map Name
- Map file name, by condition

<table>
<thead>
<tr>
<th>Map Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>map_name</td>
<td>TOW.TGA, FLW-FUTURE.TGA</td>
</tr>
</tbody>
</table>

### Composite Map Name
- Map file name, composite view

<table>
<thead>
<tr>
<th>Composite Map Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite_map_name</td>
<td>ALLAREA.TGA</td>
</tr>
</tbody>
</table>

### Hex Rgb
- Hexadecimal color value of the condition

<table>
<thead>
<tr>
<th>Hex Rgb</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex_rgb</td>
<td>C5EAF</td>
</tr>
</tbody>
</table>

### Radar Enabled
- Is radar used at all in any of the currently active conditions

<table>
<thead>
<tr>
<th>Radar Enabled</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>radar_enabled</td>
<td>Y, N</td>
</tr>
</tbody>
</table>

### Radar Zoom Enabled
- Is zooming radar used at all in any of the currently active conditions

<table>
<thead>
<tr>
<th>Radar Zoom Enabled</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>radar_zoom_enabled</td>
<td>Y, N</td>
</tr>
</tbody>
</table>

### Use Radar (Bug Only)
- Should radar be shown for this condition

<table>
<thead>
<tr>
<th>Use Radar</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>use_radar</td>
<td>Y, N</td>
</tr>
</tbody>
</table>

### Zoom Radar (Bug Only)
- Should radar be zoomed for this condition

<table>
<thead>
<tr>
<th>Zoom Radar</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>zoom_radar</td>
<td>Y, N</td>
</tr>
</tbody>
</table>

---

### Other Elements

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_name1</td>
<td>Severe Thunderstorm Warning</td>
</tr>
<tr>
<td>condition_name2</td>
<td>Severe T'Storm Warning</td>
</tr>
<tr>
<td>condition_name3</td>
<td>Thunderstorm Warning</td>
</tr>
<tr>
<td>condition_name1_l2</td>
<td>Aviso de Tormenta Electrica Severa</td>
</tr>
<tr>
<td>condition_name2_l2</td>
<td>Aviso de Tormenta Elec.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cond_rgb</td>
<td>RGB color value of the condition</td>
</tr>
<tr>
<td>phenomena</td>
<td>2 character code identifying type of condition</td>
</tr>
<tr>
<td>escalation_level</td>
<td>Precedence of the ordered condition list (lower numbers are higher priority)</td>
</tr>
<tr>
<td>category</td>
<td>3 character code indicating category assigned to condition (SWV: severe, WSV: winter, MAV: marine, HVW: hydrological, NPW: non-precipitation, TPW: tropical, C01 – C05: custom)</td>
</tr>
<tr>
<td>use_map</td>
<td>Is a map generated for this condition</td>
</tr>
<tr>
<td>zone_name1</td>
<td>(Bug Only) Zone/County name, display line 1</td>
</tr>
<tr>
<td>zone_name2</td>
<td>(Bug Only) Zone/County name, display line 2</td>
</tr>
<tr>
<td>zone_name3</td>
<td>(Bug Only) Zone/County name, display line 3</td>
</tr>
<tr>
<td>state</td>
<td>(Bug Only) Postal code or marine designation</td>
</tr>
<tr>
<td>zone</td>
<td>(Bug Only) NWS zone ID (000 indicates county)</td>
</tr>
<tr>
<td>fips</td>
<td>(Bug Only) County/marine FIPS code (00000 indicates zone)</td>
</tr>
<tr>
<td>zoom_points</td>
<td>(Bug Only) 4-field CVS string for zooming in Expio (width, height, X-position, Y-position)</td>
</tr>
<tr>
<td>future_event</td>
<td>Is the effective time of the event in the future</td>
</tr>
<tr>
<td>fmtd_etime</td>
<td>(Bug Only) Expiration time</td>
</tr>
<tr>
<td>fmtd_edate</td>
<td>(Bug Only) Expiration day of the week</td>
</tr>
<tr>
<td>fmtd_edate2</td>
<td>(Bug Only) Expiration day of the week with day part (alternative to fmtd_ampm + fmtd_edate)</td>
</tr>
</tbody>
</table>
General Anatomy of SevereWx LOGIC

There are three possible states a SevereWx LOGIC file can ever be in:

**Active and On-Air**  There are active conditions in the database, and automation rules and display preferences indicate the data should be shown currently.

**All Expired**  There are no active conditions to show because everything the system is configured to monitor has now expired. (i.e. The overall event has now passed.)

**Active but Off-Air**  There are active conditions in the database but automation rules or display preferences indicate the data should not currently be shown. At any time, automation could change the state to Active and On-Air or All Expired.

It’s vital that the LOGIC file appropriately handle each of these three states. The best approach is to wrap the entire file into an IF/ELSIF/ELSE block that tests for each of the three states:

```plaintext
[% IF (MODULECOUNT('SEVEREWX') == 0) %]
  [%# STATE: All Expired %]
  [% STOPLOOP() %]

[% ELSIF (SevereWx.size > 0) %]
  [%# STATE: Active and On-Air %]
  [% global.prev_bcast_state = 'onair' %]
  [%# Commands to show bug or crawl %]

[% ELSE %]
  [%# STATE: Active but Off-Air %]
  [% IF (global.prev_bcast_state == 'onair') %]
    [%# Commands to blank the screen %]
  [% END %]
  [% global.prev_bcast_state = 'offair' %]
  [% WAITFOR('Auto-Off', '15') %]

[% END %]
```
The IF block will stop the runlevel as soon as everything has expired. The ELSIF block is where you place the commands to show the crawl or bug. Note that we set global.prev_bcast_state to a value of onair. Finally, the ELSE block is triggered if automation takes the display off air temporarily. The global.prev_bcast_state value should be checked to know if the screen needs to be cleared. Then, the value is changed to reflect the current offair state, and a 15 second pause is inserted. The ELSE block will continue to be triggered, and the display paused for 15 seconds at a time, until automation brings the display back on or everything expires.

**Generating a Crawl Display**

To create a crawl, iterate through the array element crawl_text for each condition to return each box of text currently activated to air:

```
\_crawl\CrawlTemplate\%
 [% FOREACH cond = SevereWx %]
   [%# Current condition is in the element
   cond.forced_condition_name %]
   [% FOREACH crawl = cond.crawl_text %]
     [% crawl.text %]<Image crawl_logo>%
   [% END %]
 [% END %]
\[
```

As shown in line 7, you'll want to separate each crawl box with a graphical separator.

Note that the condition data element for a Header Message is HEAD and for a Trailer Message is TAIL. Thus, if Header and Trailer Messages are active, the size of the SevereWx array will be the number of active conditions plus two, as shown in this example:

```
SevereWx element 0  (condition: HEAD/Header Message)
  crawl_text element 0: Header Message Text

SevereWx element 1  (condition: TOW/Tornado Warning)
  crawl_text element 0: Auto-Generated Summary
  crawl_text element 1: Call-to-Action Text
  crawl_text element 2: NWS Text

SevereWx element 2  (condition: SVA/Severe T'Storm Watch)
  crawl_text element 0: Auto-Generated Summary
  crawl_text element 1: Call-to-Action Text
  crawl_text element 2: Manual Crawl Box 1
  crawl_text element 3: Manual Crawl Box 2
```

As shown in line 7, you'll want to separate each crawl box with a graphical separator.
Generating a Map/Bug Display

With the broadcast engine optimized for a bug, each element of the SevereWx array will be a county or zone under a particular condition. Thus, the size of the SevereWx array will be the sum of the number of areas affected by each condition, as shown below:

- SevereWx element 0: Tornado Warning: Alpha Co.
- SevereWx element 1: Tornado Warning: Charlie Co.
- SevereWx element 2: Severe T’Storm Warning: Alpha Co.
- SevereWx element 3: Severe T’Storm Warning: Bravo Co.
- SevereWx element 4: Severe T’Storm Warning: Charlie Co.
- SevereWx element 5: Severe T’Storm Warning: Delta Co.
- SevereWx element 6: Flash Flood Watch: Bravo Co.
- SevereWx element 7: Flash Flood Watch: Charlie Co.
- SevereWx element 8: Flash Flood Watch: Echo Co.

This differs from the crawl optimization setting that returns elements by condition, not condition and area.

There are two types of maps generated by NewsTicker which contain flood fills and polygon outlines (if configured):

**Per-Condition Maps** Each of these maps highlight the affected areas for that condition only. This lets you step through each active condition, one-by-one, showing all areas affected. If a particular zone or county is under multiple conditions, such as a Severe Thunderstorm Watch and Flash Flood Watch, it would appear in both of those per-condition maps. One map is generated for each condition configured in SevereWx and the name of the map file is passed in the \texttt{map\_name} element.

**Composite Map** This map highlights only the most significant condition for each county or zone. It can provide a quick overview of the most serious threat for each area, but can’t illustrate that a particular area is under more than one threat. Using the previous example, the composite map would only show the Severe Thunderstorm Watch for that area and not the Flash Flood Watch. Since the map is a composite of the entire ADI, only one file is generated. The name of the map file is passed in the \texttt{composite\_map\_name} element.

Note that it is necessary to prepend the correct path to the location of the map files as that is not included in the filename, as shown below.

\begin{verbatim}
1 [% FOREACH cond = SevereWx %]
2 \textbackslash \_map\MapTemplate\%
\end{verbatim}
In the above example, line 3 shows the name of the condition and line 4 the zone or county name. Lines 6-10 show the expiration time if it exists. Since the NWS doesn’t provide an expiration time for every event, such as Tropical Storm Warnings, it’s important to check whether an expiration time is being passed before assuming one is.

Line 12 references a special “cursor” file. When NewsTicker builds Per-Condition Maps, it also builds secondary files that contain a color chip for each condition. The name of the file is the three-letter condition element, followed by -$CURSOR.TGA$. For example, the color chip for a Tornado Warning would be TOW-CURSOR.TGA.

Finally, line 13 shows the condition map.

**Radar Integration and Motion Zooming**

Radar is typically incorporated into the display by reading from a Targa sequence. An Expio timeline is then fired to display the image. Similarly, radar zooming is accomplished by using the texture_scale and texture_translate data elements. These values are computed based on the configured radar and user-defined zoom points. They are formatted to be used directly as the scale and texture values in Expio’s $S$ command.

The user’s intention for including radar can be configured directly in the NewsTicker GUI. For each condition, you can choose to show the radar zoomed out, show the radar zoomed in, or not show the radar at all. Therefore, it is important that LOGIC files respect these options by consulting the following data elements:

**radar_enabled** This will be set to $Y$ if radar has not been centrally disabled by the user or automatically due to outdated imagery, and at least one of the active conditions is set to display radar (with or without
zooming). It doesn’t tell you if the current condition is set to use radar however.

**radar_zoom_enabled**  This will be set to `Y` if radar has not been centrally disabled by the user or automatically due to outdated imagery, and at least one of the active conditions is set to display zooming radar. It doesn’t tell you if the current condition is set to use zooming radar however.

**use_radar**  This will be set to `Y` if radar has not been centrally disabled by the user or automatically due to outdated imagery, and the current condition is set to display radar (with or without zooming).

**zoom_radar**  This will be set to `Y` if radar has not been centrally disabled by the user or automatically due to outdated imagery, and the current condition is set to display zooming radar.

How you choose to incorporate radar into your display will determine which of the above status fields should be queried.

The example that follows shows how to use the scale and translate values to zoom into each county or zone that is under a condition specified to use zooming.

```% ToZoom = [] %
% FOREACH item = SevereWx %
  [% IF (item.zoom_radar == 'Y') %
    [% ToZoom.push(item) %
  [% END %
% END %
% ToZoom = UNIQUEARRAYHASH(ToZoom, 'forced_zone_name') %
% FOREACH item = ToZoom %
  S\_pg\Zoom\Radar\MAIN TEXTURE SCALE\% item.texture_scale \%\% CRLF %
  S\_pg\Zoom\Radar\MAIN TEXTURE TRANSLATE\% item.texture_translate \%\% CRLF %
S\_pg\Zoom\Background\MAIN TEXTURE SCALE\% item.texture_scale \%\% CRLF %
S\_pg\Zoom\Background\MAIN TEXTURE TRANSLATE\% item.texture_translate \%\% CRLF %
S\_pg\Zoom\Foreground\MAIN TEXTURE SCALE\% item.texture_scale \%\% CRLF %
S\_pg\Zoom\Foreground\MAIN TEXTURE TRANSLATE\% item.texture_translate \%\% CRLF %
```
Lines 1-6 iterate through everything currently active and creates a new array called ToZoom of anything that is flagged for zooming (zoom_radar is set to Y). But, since the same county or zone will appear multiple times in the array if it is under multiple conditions, line 8 sends the new array through the UNIQUEARRAYHASH command to remove all duplicates. Now, ToZoom contains only one reference to each county and zone currently flagged for zooming.

The remaining lines then iterate through that array and zoom/pan to each region. Note the three sets of Expio S commands – one set moves the radar image, the second set moves the background image, and the third set moves the foreground image. The Z command on line 23 then executes all six movements, frame accurate.

**Filter Options**
The format of the data sent to the LOGIC file, among other customizations, is determined by the filter attached to that instance of the module within the runlevel. Without a filter set, the broadcast engine will assume you are airing a crawl.

**Broadcast Engine Optimization** This option tells the broadcast engine the manner in which you expect the data returned – for use with a crawl or a bug. Once you select a method, you can then further refine which types of data are to be included in the crawl, or the order to display the bug. Crawls are always presented in the same order as the Crawl tab in the NewsTicker SevereWx interface.

If you’ve selected Optimize for Bug, you have an additional option: Show only most severe phenomena per area. With this option enabled, NewsTicker will only show one condition per area – the one with the highest priority. For example, if a Tornado Warning, Severe Thunderstorm Watch, and Flash Flood Watch are active for a particular county, only the Tornado Warning will air with this option enabled.

**Display Options > Suppress all conditions when Tornado Warnings are active** While you can create rules in your Automation Profile to hide other active conditions when a tornado warning is active, this is a second, more direct option. It’s designed for stations that don’t use Automation Profiles.
Display Options > Show lower significance of the same phenomena for each area  With storm-based warnings, only a portion of a county may be under a particular condition such as a Tornado Warning, with the rest of the county under a Tornado Watch. With this option disabled, only the Tornado Warning will be shown. Enabled, the county will show both a Tornado Warning and a Tornado Watch on air.

Display Options > Hide future events  In the Configure screen of SevereWx, you can set whether you wish to display future (long-fuse) events, such as a Winter Storm Warning that doesn’t go into effect until the following day. If you have toggled this feature on, then you can override that global option on-air and not air the condition until the event start time is reached.

Display Options > Filter data set for Just Issued  When enabled, this option will watch for changes to the database and only send items to the LOGIC file if a change is detected. It is suitable for a “just issued” popup display, but not a continuous loop since data will only be sent to the LOGIC file once.

Display Options > Display condition name using  When a condition is sent to air, the field from SevereWx Edit Conditions specified here will be used. This allows you to use a long-form version in one display (such as a crawl), and an abbreviated version in another (such as a bug).

Display Options > Display county/zone name using  When an area name is sent to air, the field from SevereWx Edit Zones/Counties specified here will be used. This allows you to use a long-form version in one display (such as a crawl), and an abbreviated version in another (such as a bug).

Display Options > Automation Profile  If you’ve created an Automation Profile from the SevereWx Configure menu, and wish to apply its rules to this instance of the SevereWx module in the runlevel, select it here.

Language Options  If secondary language support has been enabled, this option lets you select the language to use. English, then Spanish will display everything in English first, then everything in Spanish. For example, let’s assume you have a Tornado Warning, Severe Thunderstorm Warning, and Flood Warning active. It would show all three in English, then show all three in Spanish. English/Spanish Mix would show the Tornado Warning in English, then in Spanish, then the Severe Thunderstorm Warning in English, then Spanish, etc.

Counties/Zones  By default, all counties and zones configured are sent to air. This setting allows you to specify a subset of areas to show.
The top depth will always be equal to or greater than the base depth. To save on-air space, you may wish to simply show it as: Base Depth: 12\" - 18\", where 12 is the base_depth and 18 is the top_depth. You will need to use an IF statement to avoid displaying 12\" - 12\".

The conditions field can vary greatly in length. It is more usable on crawls or on your Web site. If you wish to use it on a cuts display, it is probably best to use the split function in your LOGIC file to only display the first condition. For instance, if the condition is Packed Powder, Loose Granular, Machine Made, the split function would only return and display Packed Powder.

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>name1</td>
</tr>
<tr>
<td>Name of ski area, formatted for air</td>
</tr>
<tr>
<td>base_depth</td>
</tr>
<tr>
<td>Inches of snow at the base of the mountain</td>
</tr>
<tr>
<td>top_depth</td>
</tr>
<tr>
<td>Inches of snow at the top of the mountain</td>
</tr>
<tr>
<td>num_lifts_open</td>
</tr>
<tr>
<td>Number of lifts currently open</td>
</tr>
<tr>
<td>num_lifts</td>
</tr>
<tr>
<td>Total lifts, including those not open</td>
</tr>
<tr>
<td>pct_lifts_open</td>
</tr>
<tr>
<td>Percentage of lifts open</td>
</tr>
<tr>
<td>num_runs_open</td>
</tr>
<tr>
<td>Number of runs open</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ski_id (Key)</td>
</tr>
<tr>
<td>Unique ID of ski area</td>
</tr>
<tr>
<td>new_snow_24</td>
</tr>
<tr>
<td>Inches of snow received in last 24 hours</td>
</tr>
<tr>
<td>conditions</td>
</tr>
<tr>
<td>Current condition of the snow on the slope</td>
</tr>
<tr>
<td>km_XC</td>
</tr>
<tr>
<td>Kilometers of cross-country trails open</td>
</tr>
<tr>
<td>hours</td>
</tr>
<tr>
<td>Ski area’s hours of operation</td>
</tr>
<tr>
<td>url</td>
</tr>
<tr>
<td>URL of ski area’s homepage</td>
</tr>
<tr>
<td>email</td>
</tr>
<tr>
<td>General email address of ski area</td>
</tr>
<tr>
<td>updated</td>
</tr>
</tbody>
</table>
Sports

A couple times a year, there are days where no professional or college sports are being played, such as leading up to the baseball all-star series. So, if you are formatting sports scores for a crawl, you may want to wrap your LOGIC file with a statement like:

```
[% IF Sports.size > 0 %]
  Put sports LOGIC commands here
[% END %]
```

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>league_name1</td>
</tr>
<tr>
<td>League name, display line 1</td>
</tr>
<tr>
<td>league_name2</td>
</tr>
<tr>
<td>League name, display line 2</td>
</tr>
<tr>
<td>league_name3</td>
</tr>
<tr>
<td>League name, display line 3</td>
</tr>
<tr>
<td>forced_visit_team</td>
</tr>
<tr>
<td>The first field found that contains the name of the visiting team</td>
</tr>
<tr>
<td>vt_name1</td>
</tr>
<tr>
<td>Visiting team, display line 1</td>
</tr>
<tr>
<td>vt_name2</td>
</tr>
<tr>
<td>Visiting team, display line 2</td>
</tr>
<tr>
<td>vt_name3</td>
</tr>
<tr>
<td>Visiting team, display line 3</td>
</tr>
<tr>
<td>vt_winner</td>
</tr>
<tr>
<td>Y if the visiting team won the game</td>
</tr>
<tr>
<td>forced_home_team</td>
</tr>
<tr>
<td>The first field found that contains the name of the home team</td>
</tr>
<tr>
<td>ht_name1</td>
</tr>
<tr>
<td>Home team, display line 1</td>
</tr>
<tr>
<td>ht_name2</td>
</tr>
<tr>
<td>Home team, display line 2</td>
</tr>
<tr>
<td>ht_name3</td>
</tr>
<tr>
<td>Home team, display line 3</td>
</tr>
<tr>
<td>ht_winner</td>
</tr>
<tr>
<td>Y if the home team won the game</td>
</tr>
<tr>
<td>forced_period</td>
</tr>
<tr>
<td>The first field found that contains a status, or the game time otherwise</td>
</tr>
<tr>
<td>Field</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Period, display line 1</td>
</tr>
<tr>
<td>Period, display line 2</td>
</tr>
<tr>
<td>Period, display line 3</td>
</tr>
<tr>
<td>visit_score</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>home_score</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>vt_rank</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ht_rank</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>alt_home_team</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>alt_visit_team</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>alt_period</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>game_time</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>game_type</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>final</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>notes</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>notes_all</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>OTHER ELEMENTS</td>
</tr>
<tr>
<td>game_id</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>game_date</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>game_ampm</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>game_datetime</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>league</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>league name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>home_team</td>
</tr>
<tr>
<td>Unique home team identifier</td>
</tr>
<tr>
<td>visit_team</td>
</tr>
<tr>
<td>Unique visiting team identifier</td>
</tr>
<tr>
<td>period</td>
</tr>
<tr>
<td>Unique period identifier</td>
</tr>
<tr>
<td>updated</td>
</tr>
<tr>
<td>Date/time record was last updated</td>
</tr>
</tbody>
</table>

**Game/Team Notes**

The notes value will return an array up to three elements long. The order of data returned is game notes (if present), then visiting team notes (if present), followed by home team notes (again, if present). So, if there are game notes and home team notes, but not visiting team notes, the array will be two elements long with the first element being the game notes and the second element being the home team notes.

The hash that is returned has only three elements you’ll potentially care about:

- *itype*  
  A = game note, I = visit team note, 0 = home team note

- *display_line*  
  Note information, with lines separated by a carriage-return and line-feed

- *updated*  
  Date/time record was last updated

Remember that each of three notes may contain multiple lines. If your on-air display will crawl the information, you can convert the carriage-return line-feed to a crawl separator. The command below will return the note with the logo inserted:

```plaintext
[% note = game.note.0 %]
[% note.display_line.replace('
', '<Image crawl_logo_LOGO>') %]
```

If you wish to hot-change each line within a given note, you could split each on the carriage-return line-feed:

```plaintext
[% lines = game.note.0.display_line.split('
') %]
[% FOREACH line = lines %]
  Display "line" here
[% END %]
```

Whereas the notes element assumes you care that a particular note is a visiting team note, home team note, or game note, the notes_all element assumes you do **not** care. Each element in the array is an individual line...
of data. The elements are ordered the same as notes (game notes, then visiting team notes, then home team notes). Multiple lines within a given note type have also been broken out. Thus, if you entered three game notes, no visiting team notes, and two home team notes, the notes_all array would be five elements long.

To crawl all of the notes at once:

```
[% FOREACH game = Sports %]
\W\_page\_template\
    [% FOREACH note = game.notes_all %]
        [% note %]<Image crawl_logo_LOGO>
    [% END %]
\[\[CRLF %
...
```

Since the notes_all element doesn’t tell you the type of each note, it is only useful for those displays where displaying all of them in a single series is appropriate. Use notes if you wish to change the display based on the type of note.

**Filter Options**

The categories sent to the LOGIC file, and their ordering, are determined by the filter attached to that instance of the module within the runlevel. By default, all categories are sent in the order in which they appear in the GUI.

**Leagues** This option allows you to determine whether you want to show all leagues, just a select grouping, or all leagues except for specific ones. By choosing to show all leagues except for ones you designate, new leagues will automatically appear in your display as you add them.

**League Ordering** By default, leagues are sent to the LOGIC file in the order they appear in the GUI, which you can change by clicking Configure in the Sports module. You can override that by selecting Use custom sort order and arranging the leagues in your desired order.

**Game Ordering** Within a league, this option determines how the games are sorted, defaulted to Home Team Name 1. If you want to show games in action first, followed by scheduled games, when you show the league, then toggle on the Group by... checkbox. You also have the option of showing teams with a rank first, in order of rank, followed by unranked teams in the league by selecting the Show games with ranked teams first option.

**Game Filtering** This option allows you to skip games that don’t have results. This is useful when showing high school scores where not every team may have reported by the time you go to air.
Winner/Loser Highlights  These settings allow you to easily change the color of a team that has won or lost, if using Newsroom’s Freedom CG. These options will only be used if the status of the game represents a final status (F, F/OT, F/10, etc) and one team has a higher score than another. If you just use a winner color, the loser will automatically use the default font color. The value must be a six-character hexadecimal value.

Team Rankings  If you use team rankings, there are two ways to include them in the LOGIC file. You can either create a separate field in your template and feed it as its own data element, or you can simply prepend the team name with the ranking. If this option is set, and a rank is available for the team, it will automatically place that ranking before the team name, such as: (1) Michigan. To accommodate stations who want to only show teams that are ranked, such as CFB, but don’t want to toggle team visibility every week, there are two options: Include ranked teams that have been marked as invisible, and Only show teams with rankings.

Terrorism 2

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
</tr>
<tr>
<td>description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond_key (Key)</td>
</tr>
<tr>
<td>display_order</td>
</tr>
<tr>
<td>updated</td>
</tr>
</tbody>
</table>

TrafficMan 4

TrafficMan is the module for manually inserted traffic. (It will appear as Traffic on the main menu, but uses the TrafficMan LOGIC file directory.) Like with Headlines, you will only have data for the number of display lines configured (display_lines). Consistent with the GUI, each category is assigned a default icon. The user can override that icon on an incident by incident basis. But, in the event you wish to use the
default category selection as a second icon, its values are included (default_).

<table>
<thead>
<tr>
<th>COMMONLY USED ELEMENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>display_line1</td>
<td>First line of incident text</td>
</tr>
<tr>
<td>display_line2</td>
<td>Second line of incident text</td>
</tr>
<tr>
<td>display_line3</td>
<td>Third line of incident text</td>
</tr>
<tr>
<td>category_line1</td>
<td>Category name incident is associated with</td>
</tr>
<tr>
<td>icon_value</td>
<td>Keystore/location of selected icon</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTHER ELEMENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>incident_id</td>
<td>(Key) Unique ID of incident</td>
</tr>
<tr>
<td>icon_key</td>
<td>Unique (internal) ID of icon</td>
</tr>
<tr>
<td>icon_label</td>
<td>Name of icon used in the popup box</td>
</tr>
<tr>
<td>icon_hex</td>
<td>Hexadecimal value of icon color used in GUI (without preceding # symbol)</td>
</tr>
<tr>
<td>iorder</td>
<td>Order of icon in listing</td>
</tr>
<tr>
<td>aorder</td>
<td>Order of incident within the category (resets to 1 with each category)</td>
</tr>
<tr>
<td>icat</td>
<td>Unique ID of category</td>
</tr>
<tr>
<td>horder</td>
<td>Order of category in the GUI</td>
</tr>
<tr>
<td>default_icon_key</td>
<td>Unique (internal) ID of default icon assigned to category (may be overridden)</td>
</tr>
<tr>
<td>default_icon_label</td>
<td>Name of icon used in the popup box assigned to category (may be overridden)</td>
</tr>
<tr>
<td>default_icon_value</td>
<td>Keystore/location of icon assigned to category (may be overridden)</td>
</tr>
<tr>
<td>default_icon_hex</td>
<td>Hexadecimal value of icon color assigned to category (may be overridden)</td>
</tr>
<tr>
<td>default_iorder</td>
<td></td>
</tr>
<tr>
<td><strong>Order of icon in listing assigned to category (may be overridden)</strong></td>
<td>1, 3, 7</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| **display_lines**  
Number of available input lines for the incident | 1, 2, 3 |
| **max_row_length**  
Maximum number of characters assigned to each input line of an incident | 50, 100, 255 |
| **updated**  
Date/time record was last updated | 2004-03-12 00:05:10 |
This chapter contains various recipes you can use when creating your own LOGIC files. It guides you through common requests providing real-world examples you can include.

### A. Presentation Techniques and LOGIC-Writing Tips

#### A.1: Preventing Blank Fields From Terminating a Command Sequence

**PROBLEM**
You are sending several fields of data to a character generator using the CG’s \texttt{W} automation command. But, if one of the fields is blank (has no data in it), it stops the remainder of the \texttt{W} command.

**SOLUTION**
The problem is that the termination sequence of the write command (and all others, for that matter) is two double backslashes: \texttt{\\}. Within the command, each data field is sent with a trailing single backslash: \texttt{\}. For example:

\begin{verbatim}
 w\_fg\Sports\Los Angeles\1\New York\2\F\\n
\end{verbatim}

But, if the scores weren’t there, you might have:

\begin{verbatim}
 w\_fg\Sports\Los Angeles\New York\7:35\\n
\end{verbatim}
To prevent that from happening, you would simply insert a space as a placeholder:

\w\fg\Sports\Los Angeles\ New York\ 7:35\

There is one more thing to watch for. LOGIC removes excess white space from the automation file before processing it, which is all spaces, tabs, and carriage returns at the start and end of every line. That way, you can indent without sending spaces to your template. To prevent your intentional space from being unintentionally stripped out, you can also use the [% SPACE %] command, such as:

[% SPACE %]game.visit_score\

Another way to include spaces in place of missing data is to use the || short circuit operator:

[% current.cond_pic || SPACE %]

The above example will send the icon keystroke [% current.cond_pic %] if one is set, or a space [% SPACE %] if one is not. A word of caution – this method should not be used on any element that has the potential to return the number zero as a value since it will test “false” and a space will be sent instead of 0, which would be undesirable when it comes to a score, number of votes, or temperature.

A.2: Inserting a Degree Symbol

PROBLEM
You need to append a degree symbol to your display of temperatures.

SOLUTION
In most fonts, the degree symbol is located at decimal value 176. Use [% ASCII(“0176”) %] in your LOGIC file. When designing in Expio, you could hold down the Alt key while typing 0176 on your numeric keypad to generate the symbol.

In a few fonts, the symbol is located elsewhere. In that case, use the “Character Map” program built into Windows (usually in Accessories or Accessories > System Tools). Select the font and then click on the degree symbol. In the lower right toolbar, it will indicate the keystroke combination – the four-digit number is the decimal value.

A.3: Displaying the Current Date

PROBLEM
You want to display today’s date during the morning crawl but don’t want to enter it in manually every day.
SOLUTION

Use the DATESTRING command in a LOGIC file, such as in Headlines:

```logic
[% DATESTRING('Here are the headlines for %A, %B %-d ...') %]
```

The previous example would show:

```
Here are the headlines for Friday, March 3 ...
```

In a 24-hour environment, you could build a logical operation to choose when to display “Good Morning”, “Good Afternoon” or “Good Evening”:

```logic
[% hour = DATESTRING('%H') %]
[% IF hour < 12 %]
  Good Morning
[% ELSIF hour < 18 %]
  Good Afternoon
[% ELSE %]
  Good Evening
[% END %]
```

A.4: Showing a Countdown of Days until a Specific Event Occurs

PROBLEM

You want to include a countdown in your display until a certain event occurs, such as the New Year, a playoff game, etc.

SOLUTION

Use COMPARETIME to convert the difference between the target date and now into seconds, and then calculate days, hours, minutes, etc. from there.

```logic
[% difference = COMPARETIME('2008-01-01 00:00:00') %]
[% IF difference < 0 %]
  [% difference = difference * -1 %]
  [% seconds = difference % 60 %]
  [% difference = (difference - seconds) / 60 %]
  [% minutes = difference % 60 %]
  [% difference = (difference - minutes) / 60 %]
  [% hours = difference % 24 %]
  [% difference = (difference - hours) / 24 %]
  [% days = difference % 7 %]
  [% weeks = (difference - days) / 7 %]
  [% message = [] %]
  [% IF weeks > 0 %]
    [% msg = weeks _ PLURALIZE(weeks, ' week', ' weeks' %]
    [% message.push(msg) %]
  [% END %]
  [% IF days > 0 %]
    [% msg = days _ PLURALIZE(days, ' day', ' days' %]
    [% message.push(msg) %]
```
In the above example, the display would print something like 2 weeks, 1 day, 3 hours, 29 mins. The formatting is up to you.

In general, seconds isn’t usually displayed since the countdown will be up there statically.

Also note that the code begins with an IF statement making sure difference is a negative number. If it were a positive number, the date has already passed.

**A.5: Displaying Sunrise/Sunset Time for a Specific City**

**PROBLEM**

You would like to display the automated sunrise and sunset times that come with TickerWire’s Currents feed.

**SOLUTION**

You could easily display the sunrise and sunset time of each city in Currents by simply expanding the FOREACH loop:

```plaintext
[% FOREACH city = Currents %]
  w\_tg\CurrentsTemplate\%
  [% city.display_name1 %@:
  [% city.temperature %@]%]
  y\_tg\%
  [% WAITFOR('nothing', 4) %]

  w\_tg\CurrentsTemplate\%
  [% city.display_name1 %@:]
  Sunrise: [% city.sunrise %@]
  y\_tg\%
  [% WAITFOR('nothing', 4) %]
```
In the above example, the display would show the current temperature for the first city, followed by the sunrise and then sunset time. Then, it would move to the next city and display the same. If you wanted to make a dedicated sunrise/sunset display without other currents data, you would simply remove it from the block.

Given that the sunrise and sunset times in a market don’t vary widely, a more practical approach may be just to display it generically for the entire area once, and not every city:

```plaintext
[\% city = Currents.0 \%]
```

A.6: Using Tabs to Show the Next Data Module(s) to be Displayed

**PROBLEM**

You have visual tabs, like a file folder, along the top of your display and you would like to show the current module, followed by the next two modules in the display.

**SOLUTION**

The template uses three data fields (the current module, the next module, the module after that). You would perform an A/B timeline effect in the template to transition from the previous set of tabs to the next.

In your LOGIC file, you would use the RLMODULEINDEX array, in conjunction with the RLMODULEINDEX pointer, to return the name of the next modules.

```plaintext
[\% thismod = RLMODULEINDEX \%]
[\% nextmod = RLMODULEINDEX + 1 \%]
[\% nextnextmod = RLMODULEINDEX + 2 \%]
```
In the previous example, the first three lines are determining the index value (i.e. position within the array) that each of the modules fall. If RL2XMODULEINDEX is always the index of the current module, then adding 1 to that number will give you the index of the next module. Since you are using RL2XMODULES as the array, there isn’t the worry that adding one or two to that number may result in an empty element when currently displaying the last module in the runlevel.

Version 4 of NewsTicker provides for filtering, allowing you to alter what gets fed into the LOGIC file directly from the runlevel editor. This means you may have one set of headlines pulling from the “local” category within Headlines, and a second set pulling from the “state” category. To get the tab to say “Local” and “State”, set the Alias of each within the runlevel editor.

**B. Responding to Real-Time Data Values**

**B.1: Displaying “Updated” Next to Election Races that have Recently Changed**

**PROBLEM**
You want to display the word “updated” (or a similar graphical icon) next to election races that have updated in the last 15 minutes.

**SOLUTION**
Feed the race’s updated field into COMPARETIME and display an icon if the difference is less than 900 seconds (15 minutes * 60 seconds).

```
[% FOREACH race = Elections %]
  w\_fg\ElectLwr3rd\[% race.race_name1 %]
  [% upsecs = COMPARETIME(race.updated) %]
  [% IF upsecs < (15 * 60) %]
```
B.2: Using Different Weather Icons for Day and Night

**PROBLEM**
Your Forecasts and/or Currents display uses graphical weather icons and you would like to use a nighttime version for evenings.

**SOLUTION**
Create two sets of icons where the night version simply has a prefix denoting it, such as `rain.tga` and `night-rain.tga`.

For Currents, the LOGIC file would select an icon based on the current time of day:

```plaintext
[% hour = DATESTRING('%H') %]
[% FOREACH city = Currents %]
\w\_fg\CurrentsTemplate\%
[city.display_name1 %]:
[city.temperature %][ASCII("0176") %]

[% IF hour < 19 %]
c:/WxIcons/[city.cond_pic %]
[% ELSE %]
c:/WxIcons/night-[city.cond_pic %]
[% END %]

[% CRLF %]
\y\_fg\[CRLF %]
[WAITFOR('nothing', 4) %]
[% END %]
```

You could also base it on the sunrise and sunset values fed from TickerWire:

```plaintext
[% hourmin = DATESTRING('%H%M') %]
[% FOREACH city = Currents %]
\w\_fg\CurrentsTemplate\%
[city.display_name1 %]:
[city.temperature %][ASCII("0176") %]

[% srise = city.sunrise.replace(':', '') %]
[% sset = city.sunset.replace(': ', '') + 1200 %]

[% IF ((hourmin > srise) && (hourmin < sset)) %]
c:/WxIcons/[city.cond_pic %]
[% ELSE %]
c:/WxIcons/night-[city.cond_pic %]
[% END %]
```
For Forecasts, it would be based on the daypart.

```%
[FOREACH wx = Forecasts %]
  w_fg\ForecastsTemplate\%
  [wx.city_label %]
  [wx.forecast_label %]

  [% IF ((wx.forecast_label == "Tonight") ||
    (wx.forecast_label == "Tomorrow Night")) %]
    c:/WxIcons/night-% wx.pic %
  [% ELSE %]
    c:/WxIcons/% wx.pic %
  [% END %]
%
```

B.3: Trapping (Reacting To) a Blank Field

**PROBLEM**
If a particular field is blank, such as a forecast description, you want to skip over it (or do something else).

**SOLUTION**
A failsafe method is to check for the number of characters in the field within an `IF` or `UNLESS` statement, which then allows you to react, such as skipping it completely in the example below.

```%
[FOREACH wx = Forecast %]
  [% NEXT UNLESS wx.forecast.length %]
  w_wxForecast://% wx.forecast_label %]
  [% wx.forecast %]
%
```

If the `length` (number of characters in `wx.forecast`) is equal to zero, then the `NEXT` command is issued and the rest of the block is skipped. A longer way to write the same command (and get the same results) is:

```%
[FOREACH wx = Forecast %]
  [% NEXT UNLESS (wx.forecast.length > 0) %]
%
```

You can also use a double-pipe (||) as a logical OR operator to detect blank fields. In this example, the variable `teamname` checks to see which
field (vt_name1, vt_name2, etc.) is not empty, using the first one that contains a value:

```%
[FOREACH team = Sports %]
  [teamname = team.vt_name1 || team.vt_name2 ||
   team.vt_name3 || team.alt_visit_team %]
%
```

...

A word of caution – this method should not be used on any element that has the potential to return the number zero as a value since it will test. If team.vt_score is equal to 0, a valid score, note the difference in how LOGIC responds to two different sets of commands attempting to send a space if the field is empty:

```%
[team.vt_score || SPACE %] Sends a SPACE
%
[IF team.vt_score.length %]
  [team.vt_score %] Sends a 0
[ELSE %]
  [SPACE %]
[END %]
%
```

When using the || operator, always consider if 0 will ever be a valid value, such as scores, number of votes in an election, or temperature.

**B.4: Using Graphical Up and Down Arrows for Stock Values**

**PROBLEM**

Instead of using a + or - sign, you want to display a graphical arrow.

**SOLUTION**

Use the changeUpDown value as a basis for selecting the appropriate image. Keep in mind that the value could also be null (empty) if the stock is unchanged.

```%
[FOREACH stock = Finance %]
  \fg\Stocks[stock.name1 %]
  [stock.last %]
  [
    [IF stock.changeUpDown == "+" %]
      <UpArrow>
    [ELSIF stock.changeUpDown == "+" %]
      <UpArrow>
    [ELSE %]
      <Unchanged>
    [END %]
    [stock.changeAmount %][CRLF %]
  [
    \fg[stock.last %][CRLF %]
    [WAITFOR('nothing', 4) %]
  [
```
B.5: Changing the Game Winner a Different Color in Sports

PROBLEM
You want to turn the team that won the sports game a different color.

SOLUTION
You need to (1) test whether the game is over, and (2) determine which team won, if any.

The final data element should be used since a status of F or Final may not be the only status of a game that has gone final. For example, games that end in overtime could be listed as F/OT or F/13.

Font color is specified using the <color ...> command syntax in the \W statement. A hexadecimal value must be supplied, which can be obtained by using popular graphics packages like Adobe Photoshop.

The following example would turn the visiting team and score a different color if they won the game:

\[% IF (game.final == 'Y') \&\& (game.visit_score > game.home_score) %\]
\<color 0xFAFA00>\[
\[% UC(game.vt_name1) %\]
\[% SPACE %\]
\[% game.visit_score %\]</color>\[
\[% ELSE %\]
\[% UC(game.vt_name1) %\]
\[% SPACE %\]
\[% game.visit_score %\]
\[% END %\]

The same code could be easily modified for the home team as well.

Note that broadcast filters in version 4 of Sports allow you to directly specify a color value, making it easier for stations to change the color themselves. But, this recipe may have other uses such as changing the font style of a winning team.

B.6: Skipping Data Older Than a Specified Length of Time

PROBLEM
You wish to not show certain pieces of data if they haven’t been updated in awhile.

SOLUTION
Use the particular piece of data’s updated element and COMPARETIME to skip the data if it doesn’t match your particular criteria. To skip lottery games that haven’t been updated in two days:
B.7: Sending Each Lottery Number as a Graphical Ball

PROBLEM
You want to use graphical images for each number in a lottery game.

SOLUTION
Use the \texttt{split} command to separate out each number, and then build your template to show the number on top of a generic ball.

This gets a bit tricky because bonus numbers are all included in a single string. However, it always follows a predictable format of 1-2-3 (4) where each number is hyphen-separated, and the bonus number is surrounded by parentheses.

In this example, each number is sent to the template in its own data field, which the bonus number (if any) being sent last.

\begin{verbatim}
[% FOREACH game = Lottery %]
  [%# Separate regular and bonus numbers %]
  [% ndata = game.results.split(' ') %]
  [%# Put regular numbers into an array %]
  [% nums = ndata.0.split('-') %]
  [%# Remove ( and ) from bonus number %]
  [% ndata.1 = ndata.1.replace('[\(\)]', '') %]
  [% nums.push(ndata.1) %]

  w\_fg\Lottery1\
  [% FOREACH num = nums %]
  [% num %]
  [% END %]\\% CRLF %
  y\_fg\
  [% CRLF %]
  [% WAITFOR('EOT Lottery','15') %]

[% END %]
\end{verbatim}

B.8: Showing Windchill or Heat Index Instead of Actual Temperature

PROBLEM
You want to show the windchill or heat index temperature if it is different than the actual temperature.
SOLUTION
Test the value to determine if it is different – you can have it display instead of the temperature, or make it an addition to the display.

\W\_crawl\\\Crawl\\
FROM THE WEATHER CENTER, HERE ARE THE CURRENT
CONDITIONS AROUND THE AREA ...[% SPACE %]

[% FOREACH city = Currents %]
  [% UC(city.display_name1) %]: [% SPACE %]
  [% UC(city.condition_name1) %], [% SPACE %]
  [% city.temperature%][% ASCII("0176") %]

  [% IF city.windchill < city.temperature %]
    [% SPACE %]WITH A WINDCHILL FACTOR OF
    [% SPACE %][% city.windchill %]
    [% ASCII("0176") %]
  [% ELSIF city.hindex > city.temperature %]
    [% SPACE %]WITH A HEAT INDEX OF
    [% SPACE %][% city.hindex %]
    [% ASCII("0176") %]
  [% END %]

<image crawl_logo>
[% END %]
\[
Y\_crawl\[

If you want to show it in your time/temp display, you can use the CURRENTDATA directive as in:

  [% curdat = CURRENTDATA %]
  [% curdat.temperature %] (Same as [% CURRENTTEMP %])
  [% curdat.windchill %]
  [% curdat.hindex %]

B.9: Only Show Terrorism Threat Level If Above Normal

PROBLEM
You only want to display the terrorism threat level if it is orange or red.

SOLUTION
Test display_order for a value that is greater than 3, since orange is 4 and red is 5.

  [% level = Terrorism.0 %]
  [% IF level.display_order > 3 %]
    w\_tl\TerrorLevel[% level.color %]\[
    y\_tl\y\_tl\[
    [% WAITFOR(‘nothing’, ’4’) %]
  [% END %]

B.10: Dropping All Other Data Modules if More Than 10 Closings are Active

**PROBLEM**
You want closings to be part of your normal morning show runlevel. If there are just a few closings (10 or less), then it should keep everything running normally, useful for when there is a water main break just affecting a small number of organizations. But if there are more than 10, you want to only showing closings and skip the others.

**SOLUTION**
Place the Closings module first in the runlevel list and set a global variable that contains the number of active records. Then, test that variable in your other LOGIC files.

In your **ENDLOOP** block, you would record the number of active records:

```%
% global.activeclosings = CRECTOTAL %
```

At the top of all remaining LOGIC files, you would put something like:

```%
% NEXTMODULE IF global.activeclosings > 10 %
```

You may wish to put the threshold into its own global variable so you can make one change in your initialization file if you decide to raise it.

```%
% global.skipif = '10' %
# Init file
%
% NEXTMODULE IF global.activeclosings >
% global.skipif %
```

B.11: Speeding Up the Closings Display Based on Number of Active Records

**PROBLEM**
You want to sleep six seconds between each record if you have 1 – 20 organizations closed, five seconds if you have 21 – 40, and four if it is greater than 40.

**SOLUTION**
Use CRECTOTAL to determine the number of active records, put the sleep time into a global variable, and then call for it within the **RECORDFOUND** block.

Your **STARTLOOP** block might contain something like:

```%
% IF CRECTOTAL < 21 %
  % global.sleep = '6' %
% ELSIF CRECTOTAL < 41 %
  % global.sleep = '5' %
% ELSE %
  % global.sleep = '4' %
% END %
```
Then, after displaying the information in the RECORDFOUND block, you would feed global.sleep as the timeout in the WAITFOR command:

```plaintext
<% WAITFOR('nothing', global.sleep) %>
```

### C. Sorting, Filtering and Grouping

#### C.1: Showing Two Lines of Stocks at a Time

**PROBLEM**

You want to display the first stock on line 1 and the second stock on line 2, then on the next page, you want the third stock on line 1 and the fourth stock on line 2, etc.

**SOLUTION**

Use a BLOCK that, every time it is called, does a shift on that array containing all of the data elements, thus reducing it in size. Then, call for it within a WHILE loop that breaks once all elements in the array have been shown:

```plaintext
1 [% count = '0' %]
2 [% WHILE count < Finance.size %]
3   W\_stocks\Stocks\[
4       [% PROCESS nextrecord %]
5       [% PROCESS nextrecord %]
6       [% CLOCKTIME %][% CRLF %]
7       Y\_stocks[[] CRLF %]
8       [% WAITFOR('nothing', '4') %]
9     [% count = count + 2 %]
10   [% END %]
11 [% END %]
12 [% END %]
13
14 [% BLOCK nextrecord %]
15   [% rec = Finance.shift %]
16
17   [% rec.name1 %][% SPACE %]
18       [% rec.last %][% SPACE %]
19       [% rec.changeUpDown %][% rec.changeAmount %][% SPACE %]
20 [% END %]
```

Line 2 begins a WHILE loop that will test true until all of the stocks have been shown. Within the loop, it twice calls out to the nextrecord BLOCK via the PROCESS command (lines 4-5). Each time the request is made, the BLOCK shifts off the next stock to show, reducing it by one (line 15), and then outputs the automation command to display the data. As should be obvious, this display shows two stocks on-air at once.
If there are an odd number of stocks, the next record block will still be executed after the last stock for the second instance of the last page. It’s important to take this into account by feeding spaces, as shown above, so the last line will be empty.

C.2: Resorting College Sports Games Based on Team Rank

PROBLEM
You want to show the ranking of teams based on a published poll (AP Poll, Coaches Poll, etc.) but that information isn’t available in NewsTicker.

SOLUTION
In version 4 of NewsTicker Sports, poll data is available and can be sorted on using the Sports broadcast filter. In version 2, you can display the information with a minimal amount of effort. By using the second or third display field for a team, you can enter the standing there:

```
Display 1:
Michigan
Display 2:
1
Display 3:
```

Simply enter the rank number, omitting any non-digits such as parentheses. Then, in your LOGIC file, you can sort based on it:

```
[% FOREACH game = Sports %]
  [% IF game.vteam_name2 > game.hteam_name2 %]
    [% tempsort = 10 + game.vteam_name2 %]
  [% ELSIF game.vteam_name2 < game.hteam_name2 %]
    [% tempsort = 10 + game.hteam_name2 %]
  [% ELSE %]
    [% tempsort = 99 %]
  [% END %]
  [% tempsort = game.league_name1 _ tempsort _
     game.vteam_name1 %]
  [% END %]
  [% SortedSports = Sports.sort('tempsort') %]
  [% FOREACH game = SortedSports %]
    w\_sports\Sports\...
  [% END %]
```
There are two FOREACH statements – the first one is exclusively for sorting – the second one is where the actual display takes place.

Concentrating on the first one, we compare Display 2 of the visiting team with that of the home team—whichever is greatest gets recorded into the tempsort value. If neither team is ranked, then tempsort is 99 for that game.

You also note we summed 10 to the value when the team is ranked. That’s because we need tempsort to be a two-digit number since the sort is actually an alphabetical sort, not a numerical sort. If we were only dealing with rank and used Sports.nsort('tempsort') instead, then it would be numeric, but we are actually performing a secondary sort – sort first on league, then rank, then visiting team name 1 – so an alphabetical sort is required.

In an alphabetical sort, all numbers of the same digit length get sorted numerically. Thus, a ranking of 1 will actually be 11 for the purposes of sorting. This is also why unranked teams get a temporary ranking of 99 – numerically, that puts them below the rest. Keep in mind that this is just temporary – it’s just to sort on. You will correctly use the actual ranking when displaying it to air.

We then expand tempsort to be the league, followed by our modified two-digit ranking (or 99 if unranked), ending with the visiting team name. Once sorted, the games will be in order of league, with ranked teams listed first (in order of rank), followed by unranked teams in alphabetical order by visiting team name.

C.3: Resorting Sports Games Based on Status

PROBLEM
You want to change the default order of sports games within a league to show finals first, games in action second, and scheduled games third. The order of each league should be maintained as per your settings in the Sports module.

SOLUTION
Version 4 of NewsTicker Sports, using the broadcast filter, has some of these options. To do it within the LOGIC file, give each intra-league game a new sort value based on the display order and current game status (i.e. final, in-action, or scheduled).

```
[ % FOREACH game = Sports %]
[ % revdo = 100 + game.display_order %]
[ % IF game.final == 'Y' %]
```
The new sort value will be a combination of the league’s display order and a single letter to denote whether the game is final, in-action, or scheduled. The letter A will be used for finals since we want those displayed first, B for games in action, and C for scheduled games. Finally, for tertiary sorting, the actual period or game time is included. For example: 101A, 110B3rd, 104C7:30.

Logic’s sort function performs an alphabetic sort, which is needed here. Because of that, the value used for the display order (revdo above) must be represented as a consistent length. We add 100 to the display order to make sure we are always using a three-digit value. Without it, you would have 1083rd, which is display group 10, coming before 1A, which is display group 1. By adding 100 to the display group, you now have 100A coming before 110B3rd.

At the end of the recipe, we perform the actual sort based on the new sortvalue, and re-assign it back to the original Sports array making it usable in an existing LOGIC file by placing it above the main FOREACH loop handling the display of each game.

C.4: Spreading a Single Data Item over Multiple Pages

PROBLEM
You have a cuts display that switches to the next item every four seconds, and want to alternate between time and temperature in a corner of the screen. You want the time up for eight seconds, and then switch to temperature for eight seconds.

SOLUTION
Ultimately, you are leaving the time up for two iterations of the loop data, then the temperature for two iterations of the loop data. Thus, you need to keep track of whether you are showing time or temperature, and whether it has been displayed four or eight seconds.

Use a global variable to contain a number that represents the current status and increment it by one every time it is displayed. A value of 1
means it has displayed the time once. 2 means it has displayed the time twice. 3 means it has displayed the temperature once. And, 4 means it has displayed the temperature twice.

```%
global.ttdisplay = global.ttdisplay + 1%
[%
IF global.ttdisplay == 5%
[%
global.ttdisplay = 1%
[%
END%
]
%
IF global.ttdisplay < 3%
[%
CLOCKTIME%
]
%
ELSIF global.ttdisplay < 5%
[%
CURRENTTEMP%
]
[%
END%
]
```

**C.5: Showing Manually-Entered Headlines before Automated Headlines**

**PROBLEM**
You wish to change the default behavior of NewsTicker to show manual headlines before automated headlines (vs. automated before manual).

**SOLUTION**
In version 4 of NewsTicker, you can simply use a filter within the runlevel editor to determine the order data is sent to the broadcast engine.

In version 2, create two arrays, one for manual headlines and one for automated. Then, iterate through the full list of headlines, pushing each headline accordingly into the respective array based on whether **dataType** is A (automated) or M (manual). Finally, put the array back together in the desired order.

The example below would be inserted before your normal **FOREACH** loop.

```%
headsauto = []
headsmanu = []
[%
FOREACH headline = Headlines%
[%
IF headline.dataType == 'A'%
[%
headsauto.push(headline)%
[%
ELSIF headline.dataType == 'M'%
[%
headsmanu.push(headline)%
[%
END%
[%
END%
[%
Headlines = headsmanu.merge(headsauto)%
```

You could also create two LOGIC files – one that just does manual headlines, the other that just does automated headlines. That would let you choose the display order while also giving you the flexibility to drop the automated headlines out during breaking news:

```%
FOREACH headline = Headlines%
[%
NEXT UNLESS headline.dataType == 'M'%
```
C.6: Spreading Headlines throughout the Runlevel

PROBLEM
Instead of showing all headlines at once, you want to display them five at a time, such as five headlines, then current conditions, then five more headlines, then lottery results, then five more, and sports.

SOLUTION
Create a new global array (global.headcache) that contains the current headlines. Every time you show a headline from it, you shift it off the front of the array, causing it to reduce in size by one. Once you are nearing the end of the headlines, you “refill” it by pushing new ones on to the end.

This LOGIC file would be called every time you want the next five headlines to air.

There is just one thing to be careful of, and that’s multiple headlines that logically go next to each other. In other words, if headline six...
makes sense only when preceded by headline five, and you break it up, you may have some headlines that don’t stand on their own.

C.7: Showing Only a Specific Block of Headlines

**PROBLEM**
You only want to show a certain group of headlines, but others are flagged for air.

**SOLUTION**
In version 4 of NewsTicker, you can simply use a filter within the runlevel editor to determine which data is sent to the broadcast engine.

In version 2, add an IF statement to check the head_id. For example, you would do the following to only show headlines 11 – 20:

```
[% FOREACH headline = Headlines %]
  [% IF (headline.head_id >= '11') && (headline.head_id <= '20') %]
    ...
  [% END %]
[% END %]
```

Automated headlines have ID’s that are greater than 10,000, so it automatically excludes them.

D. Working with Sponsor Logos

D.1: Rotating Sponsor Logos Based on Time of Day

**PROBLEM**
You want one sponsor for your 5 AM show, a second sponsor for your 6 AM show, and a third sponsor for all other shows.

**SOLUTION**
Create a separate LOGIC file in the root directory of /web/htdocs/formats that will display the logo when called.

```
[% imgpath = 'c:/Expio Media/WZZZ/SponsorLogos/' %
[% hour = DATESTRING('%H') %]

[% IF hour < 6 %]
  [% global.sponsor = filename _ "5AM.tga" %]
[% ELSIF hour < 8 %]
  [% global.sponsor = filename _ "6AM.tga" %]
[% ELSE %]
  [% global.sponsor = filename _ "main.tga" %]
[% END %]
```
Then, in each of your individual LOGIC files, you would do something like the following within the FOREACH blocks:

```%
PROCESS '/web/htdocs/formats/sponsor.fmt'
%
```

Note that in the above code, 5AM.tga is used from midnight through 6 AM. That’s so that you can preview the sponsor before your 5 AM newscast starts.