Compose Technical Manual

Contents

Overview

Compose is a visual programming language that uses hyper-pipes between applications to allow componentised applications to be built graphically.

Various components are available to be used with the main Compose application. These components are specially designed applications in their own right. To create a program, components are dragged onto the Compose canvas where they appear as icons. They can be moved around the canvas, and links can be created between components that allow one component to transfer data to another component. Data transfer works in a very similar way to Unix pipes, except that whilst pipes are 1-dimensional, in Compose there can be many links between components. Compose applications are therefore built up by dragging links between components on the two dimensional canvas.

Once components have been linked on the Compose canvas, the whole application can be

executed as if it is one program. The Compose application then works as an intermediary to the other components to martial data and send control messages to the components.

Compose is open source software with an MIT style licence.

Licence

Introduction

This manual contains technical information about Compose that may be useful for developers who wish to create new Compose components. None of this information is needed for those that simply want to use Compose and the existing components to create applications.

Compose components communicate using wimp messages. The bulk of this manual is therefore devoted to describing these. However, there are also other details that are needed to create a component, such as the additional structure imposed on the application directory of the component.

Towards the end of the manual there is a short description of how to use the existing components coded in C as a shell for developing new components.

Although all of the current components are coded in C, any language that supports the creation of wimp applications and that allows access to the sending and receiving of wimp messages can be used to create Compose components. Thus C, BASIC and ARM assembler are all suitable languages and most other RISC OS languages should be okay too.

General information

The following details about compose have been assigned:

Application name: !Compose

Application directory: <Compose$Dir>

Message block: from &58080 (further details below)

Component application directories: <Comp*$Dir >(where * is the name of the component)

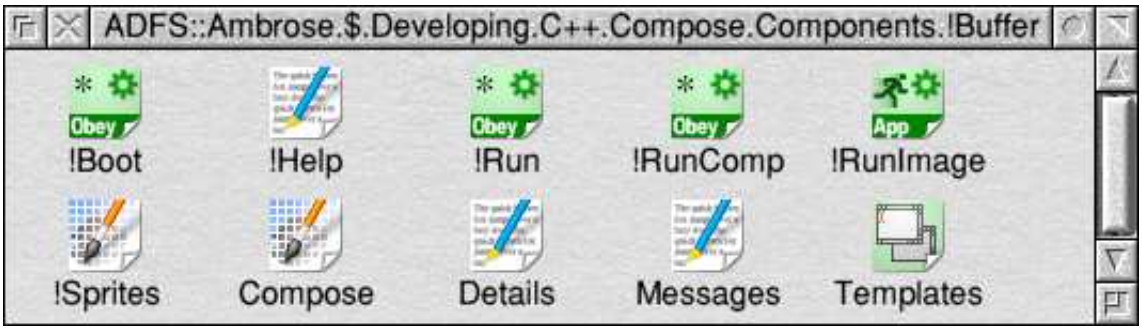
Filetype: &1bf; ComProg (further details below)

Component directory structure

Each component should be contained in a single application directory. The structure has all of the aspects of a standard application directory (*e.g.* !Boot, !Run, !Sprites, !RunImage *etc.*) along with a number of required additions.

The !RunComp file is used to launch the component and so must be included. In addition, a system variable specifying the location of the component on disc is also required (usually of the form <Comp*$Dir>), which therefore necessitates the presence of a !Boot file to set this up. The !Run file is not used by Compose, but will obviously be run if the user double clicks on the application, as with any other application directory.

The additional aspects needed are a Details text file, and a Compose sprite file, both contained in the root of the application directory. An example of the contents of a component application directory is shown below.



The contents of a component application directory

The Compose sprite file contains the image that is used when displaying the component in the main Compose canvas window. It should contain just one sprite with the name ‘compose’. There is no maximum size for this sprite, but around 100 × 100 pixels is reasonable.

The Details text file contains important information about the component. Without this file the component will not be recognised by the Compose application as being a component. When a component is dragged onto the Compose canvas, this file is read to identify and obtain details about the component. The file should be formatted along the following lines:

```
[Component]
Name : Buffer
Dir : <CompBuffer$Dir>
LinksOut : 1
LinksIn : 1

[Info]
Component : Temporarily holds data before sending it on

[Links out]
0 : Data from the buffer

[Links in]
0 : Data to be held in buffer
```

The file is initialised with a line containing only the heading [Compose]. This is directly

followed by four mandatory variable value pairs. Each on a separate line, with variable and values separated by a colon. The four variables are Name, Dir, LinksOut, and LinksIn. These stipulate the name of the component, the system variable used to identify the component's application directory, the number of output links the component has and the number of input links the component has respectively.

This is followed by two further sections. These are the [Links Out] and the [Links In] sections.

The [Links Out] section should contain one variable for each of the output links of the component. These are numbered starting at zero and the value of each must be a short line of text describing the data or information that will be output along the link.

Similarly the [Links In] section should contain one variable for each of the input links of the component. These are also numbered starting at zero and the value of each must be a short line of text describing the type of data or information that is expected to be received along the link.

Compose file format description

The Compose application allows you to save configurations of components for future use, and load them back in again at a later date. In creating a component it isn't really necessary to know how the structure of this file works, but it may none the less be of interest to component developers. This section therefore provides a very brief description of the file format, which has the registered type &1bf, or ComProg.

The Compose file format is structured using sections and variable values, in roughly human readable (text) form.

The file is comprised of sections, with each section containing a number of variables with their specified values, which can be either strings, or numbers (in text format).

Each section is delineated by the name of the section in square brackets. The variables associated with each section follow on linearly from this, separated by linefeeds. Each line should contain a variable followed by the value associated with the variable, the two being separated by a colon.

An example section might look as follow.

```
[Section title]
Var1 : 100
Var2 : Some text
Var3 : 0
```

A compose file must contain two compulsory sections, followed by a number of optional sections.

Other sections may be added at a later date, but they will conform to the above format.

The first compulsory section should have the title "Components" and should contain the variable "Num", followed by a number of pairs of optional variables with names "0D", "0P", "1D", "1P", "2D", "2P", *etc.*

The second compulsory section should have the title "Links" and should contain the variable "Num", followed by a number of optional variables with the names "0", "1", "2", *etc*.

Both of these sections may have other variables added in future versions of the file format.

The optional sections that follow should have the titles “Component 0”, “Component 1”, “Component 2”, etc. Each of these sections is associated with the configuration of a component and the names and values of the variables contained in each section will be dependent on the component in question.

Future versions of the file may change to work similarly, but based on an XML format.

An example file follows.

```
[Components]
Num : 4
0D : <CompTextView$Dir>
0P : 1073,-584
1D : <CompTextView$Dir>
1P : 1053,-358
2D : <CompServer$Dir>
2P : 666,-418
3D : <CompKeyboard$Dir>
3P : 258,-552

[Links]
Num : 3
0 : 2-0 -> 1-0
1 : 3-0 -> 0-0
2 : 3-0 -> 2-0

[Component 0]
Concat : 1
Special : 1
Mouse : 0
Warn : 0
Size : 20480

[Component 1]
Concat : 1
Special : 1
Mouse : 0
Warn : 0
Size : 20480

[Component 2]
Port : 6420

[Component 3]
Individ : 1
```

Wimp messages

The process of a component transferring or receiving data across a link makes use of wimp messages. The usual format is that a link is opened, data is transferred until the transfer is complete, possibly requiring multiple transfers and messages, and then the link is closed.

A link will often remain open for multiple wimp polls, possibly even for the entire time that a component application is running. There is no maximum or minimum length of time that a link may remain open.

Whilst a link is open, there are two ways in which data can be sent. If only a small amount

of data (224 bytes or less) needs to be sent, a Link_Send message can be used (see below). This is by far the simplest method, and avoids some of the awkward timing issues involved with using the Link_DataSave-RAMFetch-RAMTransmit message cycle discussed below. A component that wishes to send data need only support the Link_Send method if it so wishes. However, a component that wishes to receive data must support both methods for receiving data. All of this will be discussed in more detail later on.

The Compose specific wimp messages are as follows.

Message name	Message number
Link_Control	&58080
Link_Open	&58081
Link_Close	&58082
Link_Send	&58083
Link_DataSave	&58084
Link_RAMFetch	&58085
Link_RAMTransmit	&58086

We will now provide a more detailed description of each of these messages.

The following wimp messages are used by components and the Compose application to communicate with each other. For each message, two descriptions are provided. One, indicated by ‘Send’ is taken from the point of view of a component that may want to send the message. The other, indicated by ‘Receive’ is taken from the point of view of a component that may be receiving the message. The two will essentially be equivalent descriptions, just taken from opposing viewpoints. Hopefully this will serve to make clear exactly what each message should be used for.

Link_Control &58080

+20 handle
+24 action: 0=init
1=config
2=save config
3=load config
+28… further data depending on the action

Send This message is only ever sent by the main Compose application, never by a component. Therefore only receipt of this message need be implemented by components.

Receive This is a general purpose message that allows the main Compose application to communicate with components. It’s purpose depends on the value of the action word (+24).

Action 0 (init) This is sent to a component to warn it that execution is about to begin. The receiving component must take a note of the link handle (+20) and the task handle of the sending task (+4), which will be the task handle of the Compose application. All future messages must use this same link handle,

since this is a reference used by the main Compose application

Although communication appears to occur between components, in fact each component only needs to be aware of the main Compose application and the component's own input and output links. All messages are in fact sent and received only between a component and the main Compose application. The Compose application subsequently ensures that any messages are forwarded to the correct component. The component must therefore retain the task handle (+4) obtained from the init message, since all future communication must be addressed to this task and will be received from it.

Action 1 (config) This message is sent to signify that the component should open its configuration window.

Action 2 (save config)

+28 filename to *append* configuration data to (null terminated)

This signifies that a component should append its current configuration to the file indicated. The configuration must be in the following format.

```
[Component <handle>]
<var 1> : <value 1>
<var 2> : <value 2>
etc.
```

In the above, <handle> is the link handle (+20), the <var *n*>s indicate variable names (which can be any alphanumeric string) and the <value *n*>s indicate their associated values (which should also be in string format). The following is an example of this format.

```
[Component 3]
Precision : 32
Radix : 1
Size : 224
```

See the section on the Compose file format for further details.

Action 3 (load config)

+28 filename to load the configuration data from (null terminated)

The component should scan the file indicated to find the section headed by the line [Component <handle>] where <handle> is the link handle (+20). It should then read off any values it requires from this section and apply these configurations accordingly.

The component should give an error if the file does not exist. However it is *not* an error if the section does not exist within the file, or if the component cannot find a particular configuration variable that it expects.

Link_Open &58081

+20 handle
+24 link number

Send This should be sent to the main Compose application when a component wishes to open one of its output links for communication. The component should specify which link it wants to open using the link number field (+24).

Receive This is received by a component when one of its inputs links is opened by another component. The component receiving this message should ready itself for incoming data on the link, for example by assigning buffers and so on. If the link is already open, a component should either close and reopen the link, or ignore the message. It is an error if a component receives this message for a link that does not exist.

Link_Close &58082

+20 handle
+24 link number

Send This should be sent when communication on a particular link has been completed. For example, if sending a file across a link, a component would first open the link, send the file and then might close the link by sending this message to the main Compose application.

Receive When received this signifies that the data being transferred to the component across the link is complete. This may be used as a prompt for the component to take a particular action, such as processing the buffered data that was received on the link, or releasing the allocated buffer memory for the link. If a component receives a close message for a link that is not open, it may ignore it, or produce an error. It is an error if it receives such a message for a link that does not exist.

Link_Send &58083

+20 handle
+24 link number
+28 size
+32… data

Send This message is used to transfer small quantities of data across a link (less than 224 bytes). The sender should fill out the size of the data (+28) and then attach the data to the end of the message in the remaining space (+32…+256) as necessary. The data does not need to be null terminated since its size is given. Data should only be sent on a link that has previously been opened using a Link_Open message (see earlier).

If desired, it is perfectly acceptable for a component to use this method to send all of its data, as long as it can be sent in small chunks. However, if it needs to send data in larger chunks, the component should use the Link_DataSave-RAMFetch-RAMTransmit cycle discussed below. Although this is more complicated, using such a cycle is also likely to be quicker and more efficient.

Receive When received the receiving component should act on the data, for example by processing it or storing it in a buffer for later processing. If received for a link that is not open, the receiving component may error, but ideally is should act as if the link had just been opened and receive the data without error. Every component that has incoming links must be capable of accepting data in this manner.

Link_DataSave &58084

+20 handle
+24 link number
+28 size

Send This is sent by a component if it wishes to send data on a particular link that is larger than the 224 byte maximum allowed using the Link_Send message described above. The Link_DataSave message usually forms the start of a Link_DataSave-RAMFetch-RAMTransmit cycle. The size field (+28) indicates the total size of data that the component wishes to send in this particular transfer (this does *not* have to be the total amount of data sent on a link between the Link_Open and Link_Close messages, which is never explicitly specified). After sending this message, a component should expect a Link_RAMFetch message in response. If for whatever reason a component does not respond in this way, failure is likely to be silent. If a component needs to know when a Link_DataSave message fails, it should therefore send it recorded.

Receive When received this message is an indication to a component that some other component (although it may even be itself) wishes to send data to it on a particular link. If the component is able to receive the data, it should set aside some buffer space to receive the data and respond immediately with a Link_RAMFetch message (see below). Note that although the size field (+28) indicates the amount of data that will be received, the buffer does not have to be large enough to accommodate the data all in one go. If the buffer is smaller than the size of the data, several Link_RAMFetch-RAMTransmit cycles will ensue.

Link_RAMFetch &58085

+20 handle
+24 link number
+28 pointer to buffer where receiver should put data
+32 size of buffer

Send This is sent by a component in response to either a Link_DataSave, or a Link_RAMTransmit message. It can be translated to mean “ready to receive data.” The buffer pointer (+28) should be set to a pointer to a buffer that the component *sending* this message is willing to receive data into.

Receive When received by a component that wants to send data on a link, this can be interpreted as a signal from the component that sent it that it is ready to

receive data on the link. The component receiving this message will already have sent a Link_DataSave message earlier. It should respond by transferring the data it wishes to send into the buffer stipulated (+28) using a call to Wimp_TransferBlock, with the task handle from the message (+4) as the destination task for the call. Having done this, the task should respond immediately with a Link_RAMTransmit message (see below) to indicate that the data was transferred.

If the data to be transferred is larger than the size of the buffer, as much data as possible should be transferred without exceeding the size of the buffer.

If the data is larger or precisely equal to the size of the buffer, the component receiving the data will send further Link_RAMTFetch messages to obtain the remainder of the data.

Link_RAMTransmit &58086

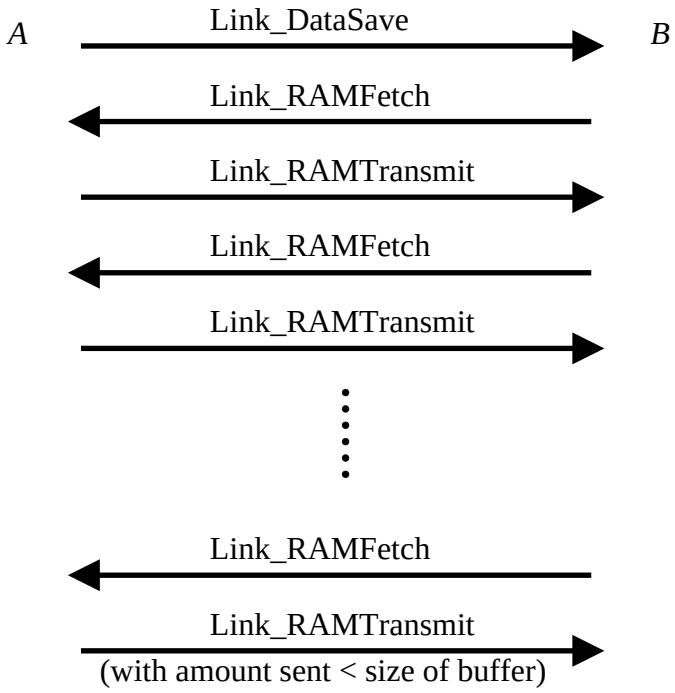
+20 handle
+24 link number
+28 pointer to buffer from RAMFetch message
+32 amount of data written

Send This is sent in response to a Link_RAMFetch message to indicate that data has been transferred into the receiver's buffer. The amount of data that has been written to the buffer should be indicated (+32). If the buffer is full, this should be set equal to the size of the buffer. If the amount sent (+32) is less than the size of the receiver's buffer (as stipulated in the previous Link_RAMFetch message), this signifies the end of the transfer. If it is equal, a further Link_RAMFetch message should be expected. In particular, if on a previous occasion the total data sent was exactly the same size as the buffer and there is no more data to send, the component should respond with a Link_RAMTransmit message with the amount transferred (+32) set to 0.

Receive When received this message means that data has been written to the component's buffer. The component should deal with the data however it needs to. If the size of the data sent (+32) is equal to the size of the component's buffer (as stipulated by an earlier Link_DataSave message) then this signifies that there is more data to be sent. In this case the component should therefore respond with a further Link_RAMFetch message (see above). As a result of this, a component should be ready to accept a Link_RAMTransmit message with the amount sent (+32) set to 0.

Transferring data

From the above we can see that a full Link_DataSave-RAMFetch-RAMTransmit transfer cycle might require messages as follows, where component *A* is sending data on a link to component *B*.



A Link_DataSave-RAMFetch-RAMTransmit transfer cycle

Note that such a transfer may take multiple wimp polls. Consequently it is not possible to send multiple Link_DataSave messages within one wimp poll and expect the ordering of the data sent to be retained. However, as a result of the way the wimp works, a component will not receive null wimp polls across a single transfer that may require multiple messages (since wimp messages take a higher priority than wimp polls). A simple way to ensure data is received in the order it was sent in is therefore to buffer data and send it only on null wimp polls.

The need to do this arises due to the nature of large data transfers, which require multiple messages to be passed between components for a single transfer of data. If Link_Send is the only means a component uses to send data, this restriction therefore does not apply. However, for larger quantities of data, the Link_DataSave-RAMFetch-RAMTransmit cycle is more efficient and its use is therefore encouraged.

Writing components using C

Writing components using C can be achieved more easily by using one of the example components provided as a template, as compared to starting from scratch. The simplest example is the Buffer component. Important declarations for this can be found in the Compose.h.General header file. There are a number of particularly useful functions that can be found in the c.Buffer source file, and these are detailed below.

Functions that can be called

The following functions can be called to achieve various results. They basically act as wrappers around the sending of messages.

void LinkSend (int nLink, char * pcData, int nSize)

Use this to send data. This is set up to use either a Link_Send message, or a Link_DataSave-RAMFetch-RAMTransmit cycle depending on the size of data that is

passed in to the function. Using this function means that the component creator does not have to worry about these more intricate aspects of the protocol.

void LinkOpen (int nLink)

Use this to open the specified link.

void LinkClose (int nLink)

Use this to close the specified link.

Placeholder functions to add code into

The following functions are set up so that they will be called when certain events occur. Primarily they will be called when an appropriate wimp message is received by the component. These functions are therefore intended as 'shells' into which code can be added to react to certain events.

void ActOnInit (void)

This is called when an init message is received. Code can be added in here to initialise the component ready for use.

void ActOnConfig (void)

This is called when a config message is received. The usual action that might be added to this function would be to open the component's configuration window.

void ActOnConfigSave (MemFile * psMemFile)

Any component-specific configuration should be saved within this function, using the MemFile routines provided. In its simplest form, a list of calls of the following form might be added:

```
SaveDetailMem (psMemFile, "var", "value");
```

void ActOnConfigLoad (MemFile * psMemFile, int nSection)

Component-specific configurations should be loaded within this function. In its simplest form, a list of calls of the following form might be added:

```
FindValueMem (psMemFile, nSection, "var", szValue, sizeof
(szValue));
```

where szValue is a buffer for a string to be stored into.

void ActOnLinkOpen (int nLink)

This will be called when a link is opened. Any code that needs executing to set up a link ready to receive data (*e.g.* the allocation of buffers) could be added here.

void ActOnLinkClose (int nLink)

This will be called when a link is closed. Any code that needs executing to tidy up a link after receiving data (*e.g.* processing the data or the deallocation of buffers) could be added here.

void ActOnLinkSend (int nLink, char * pcData, int nSize)

This will be called when a component *receives* data on a link (in spite of the function's name!). The component should do whatever needs to be done with the data, *e.g.* process it or store it in a buffer for future processing.

void ActOnLinkSent (int nLink)

This is called to signify that data has been sent. It will only be called after a call to Link_Send (see above).

Compiling the code

The code can be compiled using GCC. Obey files to compile and link each component and the main Compose application have been provided. These have been tested using GCC 3.3.3 and also require the StubsG, OSLib and UnixLib libraries.

Turing Complete

A reasonable question to ask might be "is Compose really a programming language?". In short, the answer is "Yes".

To elaborate a little further, it might be possible to argue over whether Compose constitutes a language, but it can be shown that Compose certainly does represent a form of programming. To back this up, we would want to show that Compose is Turing complete. In other words, that it has the same computation power as any other programming language has.

This claim ought to be shown by providing a formal proof that the language is Turing complete. At the moment, due to the difficulties of formalising mathematically the way components are tied together, it is not possible to provide such a proof. Instead, an example program is included in the download archive that demonstrates a Turing machine written using the Compose language. This isn't as robust as providing a formal proof, but should at least lend some credibility to the claim that the language is indeed a programming language.

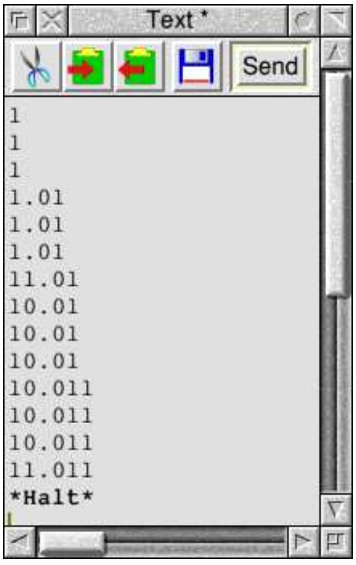
There are two example Turing machines supplied. The first, called 'Turing' demonstrates a Turing machine taken from the Wikipedia website at http://en.wikipedia.org/wiki/Turing_machine. The process for this machine is as follows.

Old state	Read symbol	Write symbol	Move	New state
s1	1	0	Right	s2
s2	1	1	Right	s2
s2	0	0	Right	s3
s3	0	1	Left	s4
s3	1	1	Right	s3
s4	1	1	Left	s4
s4	0	0	Left	s5
s5	0	1	Right	s1

The program has been set up to execute the example computation taken from the Wikipedia site, which should go along the following lines.

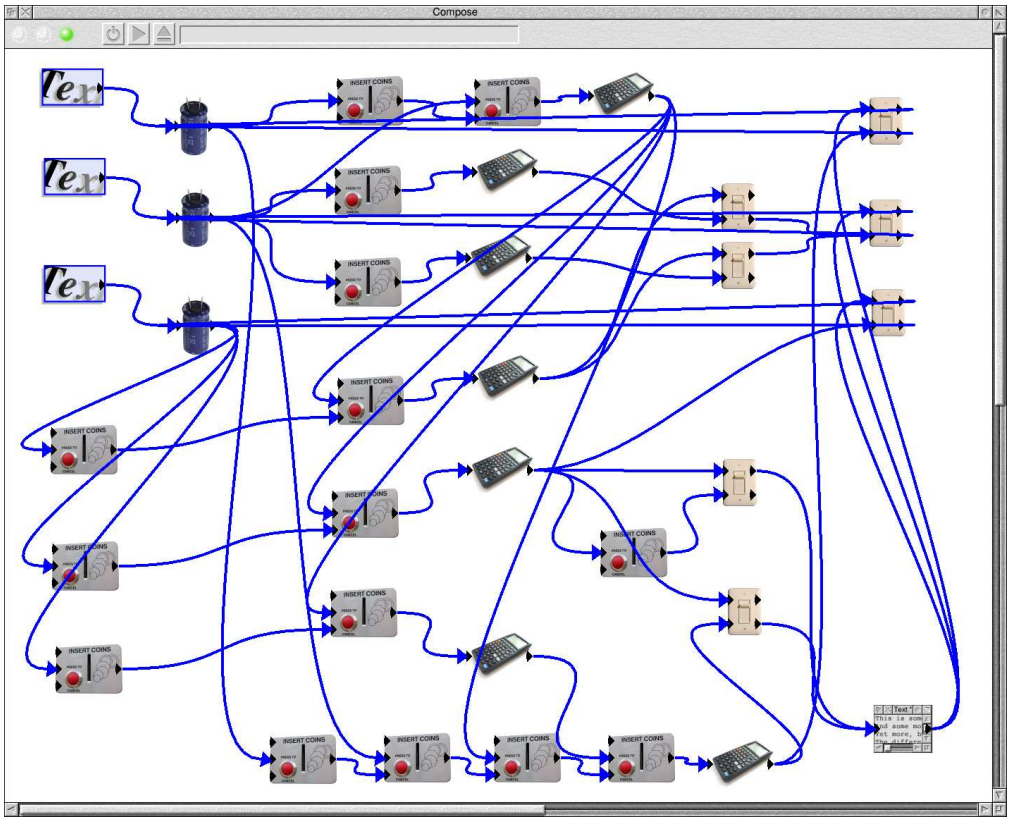
Step	State	Tape	Step	State	Tape
1	s1	**1**1	9	s2	10**0**1
2	s2	0**1**	10	s3	100**1**
3	s2	01**0**	11	s3	1001**0**
4	s3	010**0**	12	s4	100**1**1
5	s4	01**0**1	13	s4	10**0**11
6	s5	0**1**01	14	s5	1**0**011
7	s5	**0**101	15	s1	11**0**11
8	s1	1**1**01	Halt		

In the above table, the bold values in the tape column represent the current position of the machine. If you run the example Compose program you should get similar output to that shown above, as in the image below.



Output from the Turing machine ‘Turing’

Note however that since in the Compose representation it is assumed that there are infinitely many zeros at both ends of the tape, preceding or following zeros are not displayed. This sometimes makes the output from the Compose program appear different from that shown in the table. In fact they are just slightly different representations of the same thing. An image of the Compose Turing implementation is shown below. When running the Compose implementation, you can progress to the next step of the Turing machine either by selecting the 'Send' option from the TextView text window menu, or by clicking on the 'Send' button in the text window toolbar.



An implementation of a Turing machine in Compose

The second Turing machine example supplied is called 'Universal'. Unlike the other example, this actually constitutes a Universal Turing Machine, meaning that it is able to reproduce the effect of any other Turing machine depending on the values on the input tape, the start state and the start position.

The program is an implementation of a Universal Turing Machine discovered by Wolfram, as described in his book "A New Kind of Science"[1]. For more details, see the MathWorld website page concerning Turing machines at http://mathworld.wolfram.com/UniversalTuringMachine.html.

For both machines, you can change their behaviour by altering the six components along the far left hand edge of the canvas. From top to bottom, the three ConstText components represent the initial input tape, the initial position and the initial state respectively. The pattern strings of the three TokenSub components represent all of the states in which the head moves left, the new state transition formula and the new value written at the tape head formula respectively. In all of these formulas, *v1* represents the current tape contents

[1] Wolfram, S. "A New Kind of Science". Champaign, IL: Wolfram Media, p. 707.

(encoded as a floating slash number). The variable *v2* represents the tape position, where 0 is one place to the left of the decimal point increasing as the head moves left, with negative values to the right of the decimal point. The variable *v3* represents the current state of the Turing machine. The variable *v4* represents the current value underneath the machine head on the tape. Finally, the variable *v5* represents the new value that will be written underneath the machine head on the tape.