A Compendium Of Devices

Peter Seymour

14 April 2008

Preface

This is a work in progress representing some devices that can be used with the Breathe system. The aim of this book is to build a substantial description of a realistic working environment that is complete whilst remaining open. Where the need arises for new components and add-ons they can be documented here with reference to others and follow in a similar style. For instance the appendix dealing with the Basic Character Set forms a common basis for all general purpose text devices. While there is no requirement to use the same characters throughout, clearly it is beneficial to do so.

Many frameworks and libraries have been created with the goal of providing an all encompassing environment, however, they unwittingly tie themselves to existing protocols and peculiarities of other systems. While this is perhaps the most productive way to proceed in the short-term it causes a situation of constant renewal and fragility. As older less relevant systems fall out of favour so do the corresponding representations in whatever framework is being used. This leads to the temptation to start afresh often then encompassing popular features of the time and the process repeats. Worse still is the incoherency of popular implementations and the distress of having to choose one over another when what is required is an implementation of an often simple abstraction. An example is perhaps the use of databases. The programmer may have some data and the only functional requirement is for it to be persisted and then retrieved in its original state. There is much literature on different storage and retrieval techniques so a solution can be expected to exist. However, what is often found is a vast array of different configurations requiring mappings of data types, destination parameters and large numbers of so called potential "optimisations". Who demanded all these options? They must be overcome at great expense to achieve something filing cabinets have been doing for significantly longer and with more ease. Of course these options may be needed for other problems but where is the solution to this simpler problem?

This compendium has no implicit resistance to the same pitfalls but the devices outlined attempt to revert back to canonical representations that are deemed worthy by their own merits. For example the simplest text device just consumes a stream of characters. By resisting the temptation to add support for multiple character sets or control options for a fixed number of destination devices this definition should have a substantial sense of longevity. It can be argued that the choice of character set is arbitrary but that design was itself

subject to the same process. It is therefore hoped that by reapplying the same principle recursively a more coherent system evolves. Where necessary devices can be built that handle a particular piece of hardware or protocol but the first port of call should be a genuine abstraction of the core features. These can often be found by looking in more academic literature, however, it is important to select those which show usefulness in practical applications. This dual style of combining that which is fundamental with that which is useful should preserve the lifetime of these devices.

Contents

Chapter 1

Devices as an Abstraction

1.1 Introduction

The Breathe system as outlined in [1] specifies an abstract system for computing that can support multiple models. A particular model is shown in [2] which due to its historical nature in the development of the Breathe System contains a complete description of how models (U* or otherwise) interact with their contexts. The relevant sections are those on types, registers, synchronisation and signals. To bring these models into the "real" world requires a further abstraction of these contexts. Any model instance actually presents a series of computations to be performed as a single process. These computations can only interact externally in two ways: by synchronisation of external registers and by responding to signals. Any external process represents a device by providing registers to bind to and/or issuing signals. A device could therefore represent a piece of software such as a terminal window or a physical device such as a keyboard where the user is responsible for driving the process.

1.2 The Interface

When a model is described using registers and signals (as in the U*-model) it implicitly defines an interface. The interface works two ways. Firstly, those registers that are required for binding must exist externally else the model instance will be left hanging. This implies the sufficient conditions for allowing the instance to be viewed as well-posed. The second side of the interface is the set of signals that the model instance is prepared to be interrupted by. This implies a super-set of all signals that the model instance could respond to. Should a process supply a signal that is not supported then the error lies within that process and not the model instance. Each process therefore defines whether the model instance is sufficient for its purpose and can react appropriately either by not using that signal or declaring the whole system to be ill-posed.

1.3 Concurrency

The system so far outlined consists of a number of distinct processes that communicate with a single instance of a Breathe model. Care must now be taken to define the rules of communication precisely to avoid any problems common in concurrent systems such as invalid reads or writes.

1.3.1 Register Synchronisation

Communication through synchronisation of registers is a form of mutual polling. As the model instance synchronises-out to an external register it does so instantaneously. An external device reading (polling) at any time prior to the synchronisation sees the old value. Whereas at any time after it sees the new value. For completeness any read occurring at the exact instant of synchronisation can see either value. For synchronisation-in the roles are reversed but the principles remain the same. This ensures that neither process sees an invalid value.

Whilst this is an elegant solution that completely decouples the model instance from the devices it is wasteful of resources in any real implementation. Consider a clock that must update the register practically continuously whether a read is being made or not. With a little co-operation this situation can be remedied.

If the device is allowed to freeze the model instance then the path of execution may leave the model instance and be used to control the device. This does not violate the model definition since from its point of view there is no concept of real-time only the transition from one state to the next. Ideally each device ensures that under all circumstances it will return in a short period of time as it has stolen control. This does not preclude a device that can put the whole system into hibernation effectly freezing it for an indefinite period of time. Once frozen the device can perform enough operations to prepare the register values and safely update or read them atomically. In this sense it has locked all the registers it is responsible for. Notice that a deadlock situation is avoided since the model instance does not have the ability to lock anything. In the example of a clock the device only needs to operate when the current time is requested. This feels like a simple function call, an idea that will now be expanded to give a complete picture of register communication.

Under this scheme five classes of operation can be modelled: trigger, send, receive, function (a combination of a send and receive) and property (send and receive of a homogeneous type). Taking an operation f, an argument a of type A, a return value r of type R and a tag type $trig$ (size 1), then the following table summarises the available options.

Note that A or R may be structural types allowing for multiple arguments or return values and for the property $A = R$. These names can now be consistently used to describe device behaviour without explicitly mentioning low-level registers and will be used throughout this book.

Table 1.1: Operation Sheme

psuedo-code	register(s)	new syntax
$f()$	"f" $\leftarrow f : trig$	trig f
$f(a)$	"f" $\leftarrow f : A$	send $f : A$
$r = f()$	"f" $\rightarrow f : R$	recv $f : R$
$r = f(a)$	"f_{send}" $\leftarrow f : A$ "f_{recv}" $\rightarrow f : R$	func $f : A \rightarrow R$
$r = f$ $f = a$	"f" $\leftrightarrow f : A$	prop $f : A$

1.3.2 Signals

Communication in the form of signals is slightly more complicated. A simple proposal is to queue all signals in the order they occur. Potentially two or more devices could issue signals at the same instant. These can appear on the queue in any order whether it is predictable or not. When the model instance yields all accumulated signals are inserted into the yielding signal in chronological order. This ensures that each process will have its signals processed in order but no order is guaranteed between devices. This is desirable as each process is made independent of the others but internally consistent. Whilst the model instance processes these inserted signals more may arrive and are themselves queued in the recently emptied queue. This maintains the requirement that only a finite number of signals be inserted and prevents starvation whereby the model instance is constantly responding to signals.

The model definiton leaves open the possibility of using a different strategy since it requires no stronger a condition than that a finite number of signals be inserted. Any further constraints or strategies are an external concern and a master device can be used as a gateway for all other devices to communicate with the model instance. This allows for more inventive strategies that make sense in the world of the devices. For instance a 'kill' signal from any device can be pushed in front of all other signals. Communication between devices and the master device are not discussed here since they do not directly involve the model. That said the master should be an unblockable process else any device can bring down the whole system. This is easily ensured by using asynchronous communication at the device level.

For the purposes of this book a master device approach will be used. To all intents and purposes it can be thought of as part of the definition of the model. This master device process then defines the second side of the external interface and can take on various forms. One might allow the model to sit comfortably in a particular operating system responding to typical 'execute' or 'kill' signals. Another might allow the model to exist on some ad-hoc piece of hardware. All register binding by a model instance will now be with the master device which can imitate the union of all the devices it represents and even fake

others accordingly. The typical system is shown in figure 1.1.

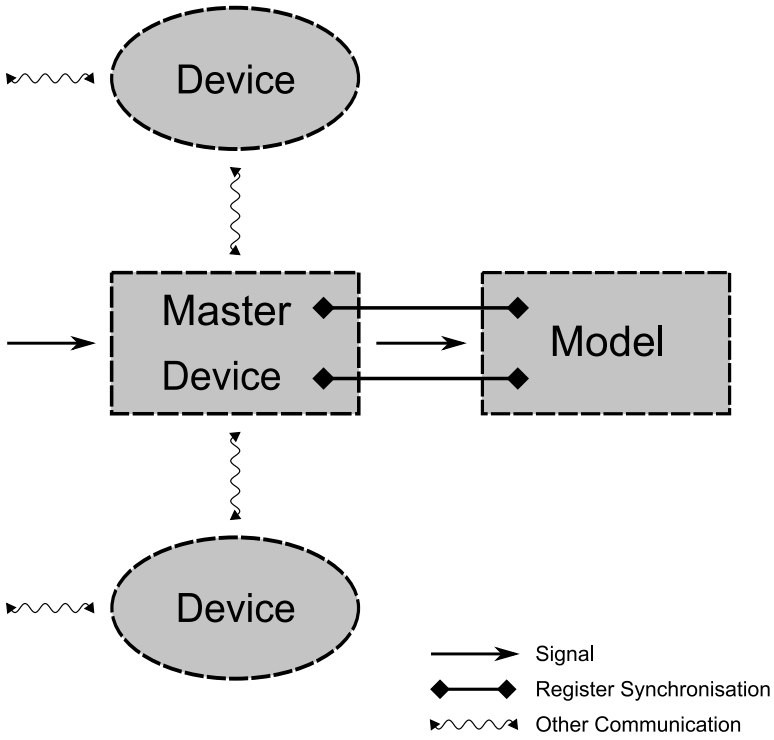


Figure 1.1: Master device based system

The alluded to 'other communication' is a hint that asynchronous communication be used. This adds a high level of fault tolerance since devices can fail safely and the master device can deal with them, say by restarting them. The master device may allow certain devices, such as the clock, to work synchronously where it trusts them or it may simply implement that functionality itself. Third party devices can be supplied at a later time and easily integrated into the system by working with the master device.

Chapter 2

Statecharts

2.1 Introduction

Statecharts were introduced by David Harel as an attempt to simplify the formal specification of state machines. Devices can conveniently be viewed as state machines where the events arise through register synchronisation. For a better introduction to statecharts please see [5] but this chapter will summarise many of the features used for device specifications. The basic principle is to view states hierarchically which prevents an explosion of states and allows re-use of common sub-systems. The method is not completely specified and allows some degree of extension. One such extension used here is the introduction of variables which allow finite state machines to cover many more devices than would otherwise be possible. Imagine a simple counter device which cannot be described using finite states without a variable.

2.1.1 State Transitions

States are depicted as rectangles with rounded corners and may have a name indicated at the top left. A transition from one state to another is labelled by an event and optionally a condition and/or action. The transition and action only take place if the condition is true otherwise no state change is observed.

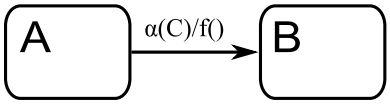


Figure 2.1: Transition example

In this example a transition occurs from state A to state B and the action $f()$ performed when event α occurs and the conditon C is true. The form of the condition is open so it can refer to states in other parts of the system, variables

or even external conditions. The action can alter the state machine or it can have some external effect.

2.1.2 State Clustering

Clustering states into larger superstates allows a system to be broken down more naturally and prevents duplication of state transitions. Any transition from a superstate applies to all substates. A transition into a superstate causes the system to choose one of the substates which is explained over the next few sections. Clustering can occur at any level.

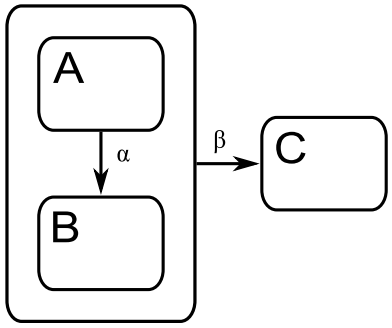


Figure 2.2: Clustering example

This example shows states A and B being clustered. When event β occurs then the system moves into state C if it was previously in state A *or* B. The event α is a usual transition from state A to B. The clustered state can be given a name and re-used in other statecharts. A transition coming into a superstate can cross the boundary and point directly at one of the substates. For instance directly from C to A.

2.1.3 Default State

A superstate may specify a default substate to be entered whenever the superstate is entered. This is shown by an arrow emanating from a dot. Any transition leading to the edge of the superstate causes the system to enter the default substate.

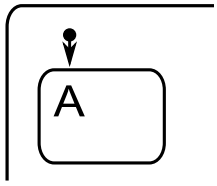


Figure 2.3: Default state example

2.1.4 State History

As an extension to the default substate is the concept of state history. Once a clustered state is exited its history *remembers* the last substate and on re-entry goes directly into that state. A default state is still required for the first entrance.

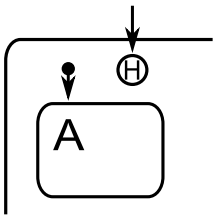


Figure 2.4: State history example

As shown the history is depicted by the letter 'H' in a circle. The first entrance to the clustered state goes to A but subsequent entrances revert to the last used substate.

2.1.5 Orthogonality

Statecharts have a powerful mechanism to avoid state explosion by allowing states to be constructed using an *orthogonal product*. By decomposing a state into orthogonal clustered states allows product combinations to be represented canonically. The following example has a superstate with left and right component denoted L and R respectively. The L component takes on a state A or B whilst the R component simultaneously takes on a state C or D. This product state transitions by looking at the transitions of each component.

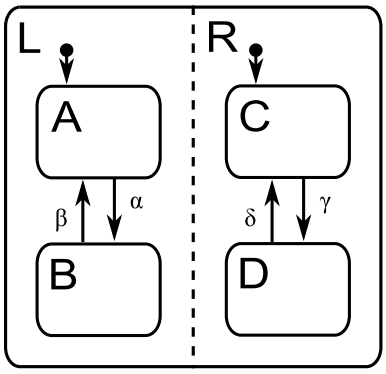


Figure 2.5: Orthogonality example

The next diagram is the equivalent state system but without using orthogonality.

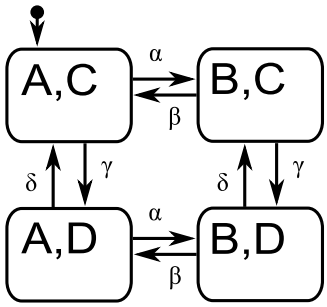


Figure 2.6: State explosion

Notice that the orthogonal state $\langle L, R \rangle$ requires a default state for each component. By using conditions that refer to orthogonal components allows for very complicated state systems to be modelled. For instance γ could be replaced by $\gamma(L\ in\ A)$ meaning the R component only changes from C to D when L is in A. A state history can be used on any component and any number of components may be used.

2.1.6 Variables

Variables have been added as an extension to the original statecharts. These are introduced as an ellipse containing the variable name and its type. The variable then has a state that can be updated using actions and referred to in conditions.

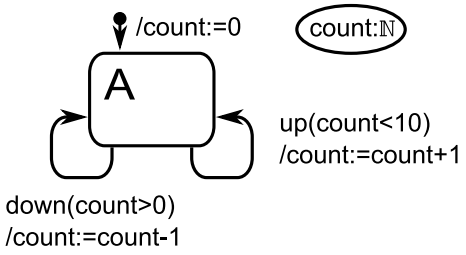


Figure 2.7: Variables example

This example shows a counter (natural number) that can be increased or decreased by one without going below zero. Initially it is set to zero.

A second use for variables is to allow information to be passed to the state machine (device) by the Breathe model. This is acheived by binding the statechart variable to one of the registers that constitute part of the device's interface. Changes to one are instantly reflected in the other.

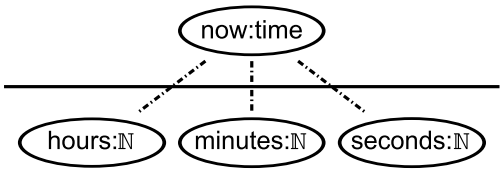


Figure 2.8: Variable binding example

Here the model register *now* of structural type *time* (composed of hours, minutes and seconds) is bound to three variables in the state machine.

Chapter 3

Describing Devices

This chapter gives a guide to describing devices and standardises much of the format. Each description should include the core features that are common across all devices and allow them to work consistently with each other. These will be displayed in a table near the beginning of the description and allow a quick overview to be found. Where names and texts are used it is assumed these are all written with the basic character set as set out in the appendix.

Devices can be logically grouped into families of similar domains. For instance output devices differ from storage devices. This grouping can be represented by chapters and sections.

Many aspects of devices can be collected together to form sub-devices. The purpose of these sub-devices is to allow easy composition of bigger devices. Such devices are referred to as utility devices.

For a clearer exposition of large scale computing systems using devices see [1] which introduces some terminology used here.

3.1 Synopsis

The synopsis introduces the device and its aims. It allows the reader to very quickly see the abstractions and familiar analogies.

3.2 Properties Table

The properties of a device are laid out in a single table and should include at least the following:

3.2.1 Class Name

The name associated with this description. All device instances are instances of a certain class and although they may have unique names of their own they share their class.

3.2.2 Usage

This will either be external, internal or utility and determines how the device will be used. External and internal devices are covered in more detail in [1]. Utility devices can be used to import groups of registers and behaviour into these device descriptions.

3.2.3 Instances/Bindings

For an external device there will be one distinct device instance for each model instance. For an internal device there will be one shared device instance with multiple copies of the register interface. The number of instances or bindings available is determined by this field. For an internal device this field has no meaning,

3.2.4 Signals

A device may issues signals to one or all of its bound model instances. The complete list of signals are listed in this field.

3.2.5 Registers

The most important part of the description is the register interface. Using the operations from Figure 1.1 is the simplest way to explain what registers are available and their intended use. To those operations are added a couple more for handling utility devices.

The *util* $f : D$ operation means to introduce all the registers of the utility device D but as a sub-device called f. There may be arbitrary levels of nesting and the full sequence of sub-device names must be used to refer to a register.

The *incl* $: D$ operation introduces a copy of the sub-device D at the top-level so the register names of D appear directly.

3.2.6 Examples

Where examples can be given then they should be listed.

3.3 Behaviour

The behaviour of a device allows a model instance to know what to expect when register synchronisation occurs and when signals may arrive. Ideally this can be clearly demonstrated by a statechart accompanying text, however, other methods may be more appropriate.

3.4 Implementation

Occasionally it is useful to include notes about particular implementation details.

Chapter 4

Common Utility Devices

The devices in this chapter are all utility devices that may be useful across many domains. Most importantly are the base classes for external and internal devices. Using these allows all devices to have some common features.

The Basic Character Set is used throughout these devices as the default method of handling text. This is denoted by the following type definition:

```
bchar = 192
```

4.1 Label

A label device represents a possibly empty sequence of read-only characters. It maintains an internal cursor to denote the current reading point which starts at the beginning of the characters. Reading a character advances the cursor unless it is at the end in which case ■ is read and the cursor does not move. There is a trigger to return the cursor to the start of the characters.

class name	label
usage	utility
signals	
registers	recv get:bchar trig reset

4.2 External Device

An external device is any device that roughly corresponds to a physical device. To avoid contention only one model instance may bind to an external device instance. For example it may be the base class of monitors, printers, operating systems and terminal windows.

class name	external device
usage	utility
signals	kill
registers	util name:label

4.3 Internal Device

An internal device is more transient than an external device and is not constrained by contention issues. It can have multiple model isntances bound to it (although each has its own copy of the registers and signal channel). For example it may be the base class of a gateway to an external device or a communication pipe.

class name	internal device
usage	utility
signals	
registers	util name:label

Chapter 5

Text Devices

This chapter deals with some common text devices.

5.1 Rill

Rill is the simplest of all text devices accepting as a single stream of characters.

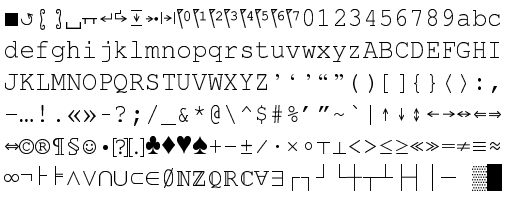
5.2 Brill

5.3 Brillio

Appendix A

The Basic Character Set

This section details the Basic Character Set. It consists of 192 character symbols as displayed below.



It is intentional that all characters have a definite symbol even if they imply some form of formatting. This allows the set to be viewed as a set of purely graphical symbols with no meaning attached if necessary. On the other hand the set is laid out it a way to form an alphabet of input symbols to a class of state machines. It is this second meaning that is discussed more fully here, however, the first use is not without its merits.

The input symbols are hierarchical with the earlier ones taking precedence over the later ones. Each symbol then may have a different meaning depending on its context, namely the symbols that precede it.

A.0.1 State Machine Control

At the top level are two control symbols which have a universal meaning across state machines and enable some form of transaction management. The first symbol is intended to perform a *commit* such that all symbols preceeding it are firmly committed to. After a commit the state machine is returned to its initial state. This sits at the top of the hierarchy since it effectively overrides all other behaviours for instance any formatting control will be reset. The second symbol *rollback* will return all behaviours to their initial state and ideally remove all symbols that were input to the machine since the last commit or start of the machine if no commits have been made. This is a very important symbol

allowing the state machine to be put into a known state before proceeding therefore removing context.

Control

00	■	commit
01	↺	rollback

When characters are used as output symbols then ■ can denote a null character.

A.0.2 Verbatim Input

At the next highest level is control for verbatim output. This permits any state machine specific processing to be overridden for all symbols occuring between closed pairs of verbatim brackets. This concept can be repeated to any level allowing input to be nested. The level of this nesting is the verbatim level. In essense this marks that a string of symbols should not be the concern of this state machine but be treated purely as a string of symbols. Perhaps this string is to be delegated to another more specialised state machine. This process allows arbitrary nesting of strings in meta-content, meta-meta-content etc.

Should the verbatim brackets become ill-formed i.e. more closing brackets are seen than opening ones then this is an error state to be handled however a machine wishes.

Since the *commit* and *rollback* symbols preceed these brackets, they will return the verbatim level to zero for the start of the next transaction.

Verbatim

02	⟦	verbbegin
03	⟧	verbend

A.0.3 Textual Formatting

Now that the state machine specific symbols have been covered, the more familiar formatting symbols appear. At this level the input symbols are to be treated perhaps as text in a document. These formatting symbols represent the usual methods to break a document down into manageable components of words, lines, paragraphs etc. How this formatting is acheived is specific to the target machine but it should be noted that there is a blank symbol near the end of the set. This can be used to achieve white space without resorting to more complicated methods.

Formatting should restore itself to an initial state after a *commit* or *rollback* symbol. When the verbatim level is greater than zero all formatting should be turned off for symbols lower in the hierarchy than the control and verbatim. Since all characters have a graphical representation this can be easily achieved.

Format

04	␣	space
05	␉	tab
06	␊	linebreak
07	␤	parabreak
08	␋	pagebreak
09	␑	sectionbreak
0a	␞	chapterbreak

A.0.4 Defined Modes

Within a formatted document there may appear complex constructs such as tags, tuples, paths through a graph etc. In fact any meta-content that is to be implied by the document. For this purpose 8 mode symbols are included. They have no pre-defined intention but allow a state machine to derive its own context sensitive information. For instance to enclose pure symbols and denote them as meta-tags, to act as a delimiter between symbol strings or even to encode higher level formating such as fonts and colours. If 8 modes are not sufficient then they can be combined into sequences to give any number of modes.

Whatever context is created should also obey the rules of the *commit* and *rollback* symbols. When the verbatim level is greater than zero then it is suggested that this context be suppressed.

Mode

0b	␀	mode0
0c	␁	mode1
0d	␂	mode2
0e	␃	mode3
0f	␄	mode4
10	␅	mode5
11	␆	mode6
12	␇	mode7

A.0.5 Pure Symbols

The following sections list the pure symbols. They are divided up into similar groups but from a state machine perspective have no meaning.

The standard digits for base 10 arithmetic.

Digits

13	0	0
⋮	⋮	⋮
1c	9	9

The lowercase letters in alphabetical order. Note that the digits followed by the first six lowercase letters lend themselves to hexadecimal numbering.

Lowercase

1d	a	a
⋮	⋮	⋮
36	z	z

The corresponding uppercase letters.

Uppercase

37	A	A
⋮	⋮	⋮
50	Z	Z

All standard forms of punctuation have been admitted. Care should be taken to be explicit about use. Whereas the dash and hyphen may be interchanged in some systems this is not the case here. The same is true of the apostrophe and single right quote or indeed the distinction between left and right quotes. The slash is distinct from its division counterpart.

Punctuation

51	'	apostrophe
52	‘	singlequoteleft
53	’	singlequoteright
54	“	doublequoteleft
55	”	doublequoteright
56	(	parenleft
57	)	parenright
58	[	squarebracketleft
59	]	squarebracketright
5a	{	braceleft
5b	}	braceright
5c	⟨	anglebracketleft
5d	⟩	anglebracketright
5e	:	colon
5f	,	comma
60	–	dash
61	…	ellipsis
62	!	exclaim
63	.	fullstop
64	«	guillemetleft
65	»	guillemetright
66	-	hyphen
67	?	questionmark
68	;	semicolon
69	/	slash

The text symbols are a collection of widely used symbols to enrich text. They include most symbols that are encountered in the field of computing for back-

ward compatability. The illegible character denotes reporting a source character that could not be read. The unprintable character on the other hand denotes a character that is known but unavailable within this character set.

Symbol		
6a	_	underscore
6b	&	ampersand
6c	*	asterisk
6d	@	at
6e	\	backslash
6f	^	circumflex
70	$	dollar
71	#	numbersign
72	%	percent
73	′	prime
74	″	doubleprime
75	~	tilde
76	`	grave
77	|	vertbar
78	↑	arrowup
79	↓	arrowdown
7a	↕	arrowupdown
7b	←	arrowleft
7c	→	arrowright
7d	↔	arrowleftright
7e	⇐	doublearrowleft
7f	⇒	doublearrowright
80	⇔	doublearrowleftright
81	©	copyright
82	®	registered
83	¶	pilcrow
84	§	sectionsign
85	☺	smiley
86	•	bullet
87	[?]	illegiblecharacter
88	[.]	unprintablecharacter
89	♣	cardclub
8a	♦	carddiamond
8b	♥	cardheart
8c	♠	cardspade

The mathematical symbols enlarge on some of the preceeding symbols to aid the clear exposition of arithmetic, logic amd set theory. Each of which have useful notations for computing related problems. Symbols such as arrows can be found earlier in the set.

Math

8d	+	plus
8e	−	minus
8f	±	plusminus
90	⁄	divisionslash
91	·	dotproduct
92	×	cartesianproduct
93	∘	composition
94	⊤	topelement
95	⊥	bottomelement
96	<	lessthan
97	>	greaterthan
98	≤	lessthanequal
99	≥	greaterthanequal
9a	≪	muchless
9b	≫	muchmore
9c	=	equal
9d	≠	notequal
9e	≡	identical
9f	≈	almostequal
a0	∞	infinity
a1	¬	notsign
a2	⊢	proves
a3	⊨	models
a4	∧	conjunction
a5	∨	disjunction
a6	∩	intersection
a7	∪	union
a8	⊂	subset
a9	∈	element
aa	∅	emptyset
ab	ℕ	naturals
ac	ℤ	integers
ad	ℚ	rationals
ae	ℝ	reals
af	ℂ	complexfield
b0	∀	forall
b1	∃	exists

The box symbols allow primitive formatting of boxes to produce tables etc. In a monospace environment this can significantly aid readability without adding too much complexity.

Box

b2	┌	boxtopleft
b3	┐	boxtopright
b4	┘	boxbottomright
b5	└	boxbottomleft
b6	┼	boxcross
b7	┬	boxuppertsection
b8	┴	boxlowertsection
b9	├	boxlefttsection
ba	┤	boxrigthtsection
bb	│	boxvertical
bc	─	boxhorizontal
bd		boxblankblock
be	▒	boxlightblock
bf	▉	boxfilledblock

Bibliography

[1] "The Breathe System", P. Seymour, 2008.

[2] "The U*-model", P. Seymour, 2008.

[3] "A Pattern For Device Drivers", P. Seymour, 2007.

[4] "Specifying Asynchronous Device Behaviour", P. Seymour, 2007.

[5] "Statecharts: A Visual Formalism For Complex Systems", D. Harel, 1987.