FORMAL SPECIFICATION
AND ANIMATION OF A
WATER LEVEL MONITORING SYSTEM

(AECB Project No. 2.223.1)
FORMAL SPECIFICATION
AND ANIMATION OF A
WATER LEVEL MONITORING SYSTEM

(AECB Project No. 2.223.1)

by

P.S. Jackson
and
P.A. Stokes

Rolls Royce and Associates Ltd.

A research report prepared for
the Atomic Energy Control Board
Ottawa, Canada

March 1993
FORMAL SPECIFICATION AND ANIMATION
OF A WATER LEVEL MONITORING SYSTEM


ABSTRACT

This report describes the Vienna Development Method (VDM) which is a formal method for software specification and development. VDM evolved out of attempts to use mathematics in programming language specifications in order to avoid ambiguities in specifications written in natural language. This report also describes the use of VDM for a real-time application, where it is used to formally specify the requirements of a water level monitoring system. The procedures and techniques used to produce an executable form (animation) of the specification are covered.

RESUMÉ

Le présent rapport décrit la "Vienna Development Method" (VDM), une méthode formelle pour l'établissement des spécifications et le développement de logiciels. La méthode résulte de tentatives d'utilisation des mathématiques pour le développement de spécifications du logiciel afin d'éviter toute ambiguïté dans la rédaction des spécifications en langage naturel. Le rapport décrit aussi l'utilisation de la méthode pour une application en temps réel où l'on définit formellement les exigences d'un système de contrôle du niveau de l'eau. Le document aborde également les procédures et les techniques de production d'une forme exécutable (animation) des spécifications.

DISCLAIMER

The Atomic Energy Control Board is not responsible for the accuracy of the statements made or opinions expressed in this publication and neither the Board nor the authors assume liability with respect to any damage or loss incurred as a result of the use made of the information contained in this publication.
TABLE OF CONTENTS

ABSTRACT ................................................................. i
1. INTRODUCTION....................................................... 1
2. VDM SPECIFICATION ............................................... 1
3. ANIMATION............................................................. 10
4. VERIFICATION......................................................... 12
5. FUTURE WORK......................................................... 16
6. VDM AND OTHER METHODS ........................................ 17
7. CONCLUSIONS.......................................................... 18
8. REFERENCES............................................................ 18

APPENDICES:

ANALYSIS................................................................. APPENDIX A
VDM SPECIFICATION.................................................... APPENDIX B
PASCAL ANIMATION.................................................... APPENDIX C
EXAMPLE ANIMATION RUN............................................. APPENDIX D
GLOSSARY................................................................. APPENDIX E
1. INTRODUCTION

One of the crucial stages in the development of any system is that of specification. Any errors introduced at this stage can be very costly to correct later on. In order to achieve precision, a specification should be written in a language which has a formal basis, and in such a way that it can be tested. At the same time, the specification should say what is to be done without introducing any unnecessary implementation detail, i.e. how it is to be done. It should also be possible to verify that a particular implementation correctly meets its specification. If the aim is to build a safety-critical system then, ideally, the method used for producing the specification should, in addition, be mature, easy to use, and well-supported (i.e. in terms of tools).

This report looks at one particular specification and development method, the Vienna Development Method, VDM. This is a model-based formal method with many of the desirable features identified above. To illustrate the principal features of VDM, the report demonstrates how VDM may be used to specify and test the requirements of a real-time Water Level Monitoring System, WLMS (see reference [8.11]). The report also describes the use of an animation in testing the VDM specification. The animation is a sequential program, written in VAX-Pascal, which provides an executable model of the VDM specification. Section 2 describes the stages involved in producing the VDM specification. Section 3 describes the procedures for producing the source code for the animation, and section 4 describes the tests that were performed using the animation. In section 5, possible areas of future work are identified.

1.1 Objectives

The main objective of this work is to demonstrate the feasibility of applying VDM to a simplified, but representative real-time control system in which precision is essential. The intention is not to use VDM to develop a final implementation, but to stop at the specification stage. Part of the work aims to show how an animation can be produced from a VDM specification, and how this can be used to test the specification. VDM has been chosen for this study because it is a mature method, it is representative of other state-based formal methods, and it has been used in relevant areas, i.e. safety-critical reactor protection control systems. Another reason for choosing VDM is the richness of the language. This has allowed VDM to be used in programming language specification.

2. VDM SPECIFICATION

This section describes the approach that has been taken in producing the VDM specification.
2.1 Background

The Vienna Development Method, VDM, has evolved out of attempts to use mathematics, rather than, say, English, to specify programming languages. The aim has been to establish a technique which could remove ambiguities from programming language specifications, and so enable the development of correct compilers. Earlier formal attempts at defining programming languages can be traced back to the pioneering work in classifying natural language by Noam Chomsky from the field of linguistics. Based on Chomsky's work, Backus and Naur were able to provide a formal definition of the syntax of Algol 60. The Backus-Naur notation, BNF, or variations of it, has since become widely used for defining the syntax of many programming languages. The success of this formal approach prompted researchers to find formal ways of defining the semantics of programming languages. An important contribution to this problem was made by McCarthy. He showed how it was possible to define the semantics of a state based programming language in terms of a hypothetical state-machine. The program research group at Oxford led by Strachey built on this work, and established a technique, known as denotational semantics, for defining the semantics of programming languages. The elegant notation of the technique is due mainly to Strachey, and the work to provide it with a well-found mathematical basis is due to Dana Scott.

Attempts to apply the technique to practical state-based languages by various researchers (e.g. Bjoerner and Jones) have led directly to the development of VDM.

To date VDM has been used successfully to define the denotational semantics of a variety of programming languages e.g. CHILL, Modula 2, Ada etc. However, the usefulness of VDM is not restricted to programming language specification; it can also be used to specify and design programs written in a programming language. This has, indeed, led to the more general use of VDM in software specification and design, such as databases, reactor protection software and operating systems.

VDM comprises both a meta-language for writing specifications (called Meta IV - note the pun), together with a framework for developing programs from specifications. Meta IV allows specifications to be written either explicitly or implicitly. An implicit specification states what is to be computed without biasing its implementation, whereas an explicit specification defines how the computation is to be done (see chapter 3 of reference [8.2]). The idea of the VDM method is to develop programs in a top-down manner. At the top level, specifications should be written as abstractly as possible; the specifications should use mathematically orientated data types (e.g. sets and mappings), and the functions and operations that manipulate them should be defined implicitly (i.e. using pre and post conditions). Specifications are then systematically refined, and at each refinement step further implementation detail is added. There are two ways of adding more implementation detail:

(1) Data refinement
(2) Operational Decomposition
With data refinement, the more abstract data structures are translated into more implementation orientated structures, e.g. sets into binary trees, or binary trees into arrays. With operational decomposition the more abstract implicit specifications are replaced by explicit specifications. Associated with each refinement step are a number of proof obligations. These can be used to verify a given stage against its predecessor.

2.2 Meta IV

The VDM meta-language is extremely large, and only a small subset of the language has been used in this study. The principal features of the language that have been used are briefly discussed here.

2.2.1 Types

VDM supports a large number of types. The types used in this study include some basic types (e.g. integer, boolean, character), enumeration types, record types, sequences (for input and output) and mappings. A mapping type is the set of all possible functions over some given domain and range. It is not a basic type, since it depends on the types of the particular domain and range used. For example, the mapping type screenType, has been used to model the CRT display. The domain of screenType is a type which models an arbitrary screen coordinate, and the range of screenType is the set of characters. A particular display is defined by selecting the appropriate value (function) of type screenType. For further details on the mapping notation see chapter 6 of reference [8.2].

It is possible to apply constraints to any VDM type by means of an invariant. An invariant on a type is a boolean expression which must always be true for any variable of this type. For example, in Appendix B, we define a type "Byte" as the set of integers subject to the constraint that any element of "Byte" must lie between 0 and 255, i.e.

\[
\text{inv}(x) = (0 \leq x) \text{ and } (x \leq 255)
\]

2.2.2 Operations

Operations are the basic building-blocks of a VDM specification; they are like functions, except that operations are allowed to read from or write to a "state" (see chapter 4 of reference [8.2]). The "state" of an operation is just the collection of external variables which it can access and change. Thus a function always returns the same result for the same parameters, whereas an operation may both depend on and alter the "state". An operation can be defined in terms of its initial "state" and final "state". Moreover, operations can either be expressed implicitly (i.e. in a way that defines what is to be computed) or explicitly (i.e. how it is to be computed). An implicit definition is more abstract, however, than an explicit definition; and most of the VDM specification has been defined implicitly. An implicitly defined operation has four parts:
(1) Signature. This consists of the operation name and any parameters it takes.

(2) External Access. This is indicated by the keyword "ext". It defines the "state" variables which the operation can access. The access can be read only (indicated by the keyword "rd"), or both read and write (indicated by the keyword "wr").

(3) Pre-condition. This is indicated by the keyword "pre". The pre-condition is a boolean function of the input parameters and the initial "state". It defines the conditions for which the operation is defined to have an effect. If for some input parameter values the pre-condition is FALSE, then the behaviour of the operation (i.e. its final "state") is undefined for these values. The operation should not be used in any situation in which this can happen.

(4) Post-condition. This is indicated by the keyword "post". This defines how the "state" is affected by the operation; and it defines the values of any exported parameters. The post condition is only valid provided the pre-condition was initially true. In the post condition it is necessary to be able to distinguish between the value of the "state" initially and its final value. To distinguish between the two, variables in the initial "state" are shown with a horizontal line above them.

An explicitly defined operation is similar to an implicitly defined operation except that the post condition is replaced by a VDM statement, i.e. a CASE statement, a LOOP statement, a BLOCK statement etc.

2.2.3 Functions

In VDM, functions cannot have side-effects, i.e. they cannot access the "state". Apart from this distinction, they resemble operations; they can be both implicitly and explicitly defined. For further details on the difference between implicit and explicit functions refer to chapter 3 of reference [8.2].

2.2.4 Expressions

In VDM, expressions can be built up from variables and functions. In addition to any user-defined functions, there are a large number of pre-defined ones. The standard functions include infix operators, e.g. "+", "*", "div" etc., and prefix operators, e.g. "hd" and "tl" (for extracting the head and tail of a list respectively) etc. Definitions for these operators can be found in reference [8.2]. Amongst these standard functions are a class of functions for constructing variables of record type from their components. These are called "make" functions, and are shown by prefacing "mk-" to the name of the record type.
VDM also has a facility, the let-expression, for providing an expression with a name. The name of a named-expression can then be used instead of the full expression. For example

\[
\text{let } t = (a + b) \text{ in } \\
x = 2 * t
\]

is equivalent to

\[
x = 2 * (a + b)
\]

VDM allows any number of let-expressions to be nested.

2.2.5 Modules

A VDM specification is a sequence of one or more modules. Each module is itself a collection of VDM definitions, e.g. constants, types, functions and operations. A module can contain an optional interface section which allows definitions to be imported or exported between modules. This provides a convenient structuring facility for large documents.

2.3 Modelling Requirements in Meta IV

As we have seen, VDM is a denotationally based method for defining the semantics of state-based programming languages. Unfortunately, we cannot use VDM to handle programming languages which are not state-based. This limitation has in fact been recognised by the VDM community at large; and it has led to attempts to extend the VDM language to support concurrency, e.g. recent work by Jones (see reference [8.3]). Another point to bear in mind is that all functions written in VDM are computable. This means that, theoretically, it should be possible to find an algorithm for implementing a VDM specification. On the other hand, not all functions are computable. This stems from the fact that the set of all computable functions is denumerable whereas, for instance, just the set of all functions over the integers is nondenumerable. This means that the overwhelming majority of possible functions cannot be written in terms of VDM. While this restriction on VDM may be essential for defining programming languages, it does limit the scope of VDM. Despite these limitations, it is still possible to use VDM to model real-time requirements, and in this section we describe the approach that has been taken.

2.3.1 System State

Although, strictly speaking, VDM is a denotationally based method, we may very loosely interpret a VDM specification in terms of a hypothetical machine which "executes" the specification. The state of the machine, at any particular moment, is defined by the values stored in the machine's "memory", i.e. the values of the VDM state variables. An obvious way to represent the system state variables in reference [8.1], (both monitored and controlled) is to use VDM state variables. Unfortunately, the representation is not exact because the state variables given in reference
[8.1] are piece-wise continuous functions of time, and VDM does not have any concept of time. The solution adopted was to model continuous functions of time discretely. So, for example, time is represented as an ascending sequence, timeSeq, of time values, i.e.

\[
\text{timeSeq} = \{t_0, t_1, t_2, \ldots \}
\]

where \( t_0 < t_1 < t_2 \ldots \)

The monitored and controlled state variables are also represented by sequences in such a way that successive sequence values correspond to successive times. For example, the value of the reset button at time \( t_i = \text{timeSeq}(i) \) is represented by \( \text{reset}(i) \) where \( \text{reset} \) is a sequence of "on" or "off" values.

To define the real-time behaviour of the system, the hypothetical machine repeatedly "reads in" the current time and the current values of the monitored variables, and "writes out" the corresponding values of the controlled state variables. In terms of VDM, a read is shown by removing the head of a sequence; a write is indicated by appending a value to a sequence. It should be remembered that since the machine is hypothetical, it is not limited by any physical laws, so the whole process is instantaneous.

In a VDM document, the state variables are global to all the operations in the document. It is generally considered to be good practice to minimise the number of these variables, especially where abstraction is required. This practice tends to produce a specification which has a hierarchical structure, and ensures that intermediate calculations are hidden at the top level. It would be possible to restrict the VDM state to just those variables that model system input and output. However, for simplicity, and for stylistic reasons, - four variables were used in the VDM specification:

- (1) monitoredInputs
- (2) systemOutput
- (3) storedData
- (4) storedVariables

The VDM variable monitoredInputs models all the monitored inputs of the WLMS, including time; whereas systemOutput models all the controlled state variables. The variables storedData and storedVariables are used as stores and do not represent any of the state variables given in reference [8.1]. The store storedData records information about the mode of operation of the WLMS, and certain timing information, e.g. the length of time the reset button has been pressed. The storedVariables store records the values of various controlled variables for the previous time step.

2.3.2 Modelling of Event Tables

Reference [8.1] uses Event Tables to define the behaviour of controlled state variables. These show how the value of a controlled state variable
changes when an event occurs. Experience with these particular tables suggests that the task of transforming Event Tables, in general, into VDM should be fairly straightforward. However, there are two main problem areas that need to be tackled:

1. concurrency
2. the exact modelling of an event

Reference [8.1] contains several Event Tables, and these are understood to define concurrent behaviour. The fact that VDM does not directly support concurrency means that this needs to be considered further. Modelling concurrency only presents a problem when separate concurrent processes can alter the same state variable e.g. when there is feedback. Fortunately, in this application, the controlled state variables defined by different Event Tables do not interfere. This means that the concurrent behaviour of the whole WLMS can be modelled by combining together the defined behaviour of the individual Event Tables. The actual VDM model achieves this using a non-terminating DO loop. The body of the loop is a single VDM operation. This operation defines a single "step" of the hypothetical state machine in which all the system inputs are read once and all the system outputs are updated once. Further, the operation is defined implicitly using VDM pre and post conditions. In this way we can avoid making any assumptions about the order in which any particular system task is performed. Taken together these provide a satisfactory model for concurrency.

The rigorous definition of a triggered event given in reference [8.1] cannot be modelled exactly in VDM. One reason for this is that VDM has no notion of time. Another reason is more practical; it is unwieldy to try to model a triggered event as a set of piecewise continuous functions. The solution adopted was to use boolean expressions to model triggered events together with enumerated types to model the changes. This approach means that we need to consider separately both the case when a triggered event occurs and when it does not occur. This is illustrated in the example below.

**Illustrative Example**

Suppose that the controlled state variable, %High Window%, is required to be set to $on$ whenever an event in event class, E, occurs where

\[ E = @T(!level High!) WHEN \\
[Inmode(*operating*) & Inmode(*allok*)] \]

The notation used above is described in detail in reference [8.1]. The VDM model for this behaviour uses the enumerated type, windowType = {on, off}, to represent values of the high window display, and the enumerated types

1. operatingType = {operating, shutdown, standby, test}
2. failureType = {allok, badlevdev, hardfail}
to model the system modes. Each triggered event from event class, E, is modelled in VDM using the boolean expression

\[(\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allok}) \land \text{ishigh} \Rightarrow (\text{highWindow} = \text{on})\]

where opmode is the current operating mode, failmode is the current failure mode, and ishigh is true when the trigger occurs.

To cater for the case when the trigger has not occurred, i.e. when ishigh is false, an additional condition is needed:

\[(\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allok}) \land \text{NOT ishigh} \Rightarrow (\text{highWindow} = \text{oldHighWindow})\]

where oldHighWindow is the previous state of the high window.

### 2.3.3 Simplifications

A number of simplifications have been made to the VDM model. These are of two main types: under-specification and abstraction. Under-specification refers to those areas of the VDM model which have been left undefined, e.g. the details for initialising the watchdog timer, and the fixed background of the CRT display. Abstraction has been used to remove some implementation details from the model. An example of this is the model for ringing the alarm. The alarm rings for 0.5s every time it receives a bell character. The alarm can be made to ring continuously by sending it a stream of bell characters at a frequency of at least one character every 0.5s. This mechanism for ringing the alarm has not been included in the model.

### 2.3.4 Limitations

There are several limitations in the present VDM model. For example, it is not possible to define rates-of-change in VDM. Moreover, although it is possible to define the tolerances in a system using VDM, it results in a specification which is nondeterministic (and difficult to animate). Consequently, the model does not consider tolerances on acceptable performance, or rates of change. If, however, it becomes essential to be able to model acceptable behaviour, then one approach might be to add tolerances to the times at which events occur. For example, if the system should respond to an event when the time exceeds 3s, then we could allow the system to respond after 3s +/- tol where "tol" is the tolerance. The actual value of "tol" would be left undefined, although the range of values it could take would be defined. Such a specification would, however, be nondeterministic, and this would need to be tackled at the animation stage. One simple way of removing the nondeterminism would be to set tol = 0.
2.4 Structuring the VDM

VDM is a tool for defining detailed semantics, and it can be difficult to produce a well-structured specification from a large number of detailed requirements. The problem can be tackled by adopting one of the structured analysis techniques found in software engineering. For the Water Level Monitoring system in reference [8.1], a simplified Yourdon-style analysis was performed. This method was chosen because it is graphical, easy to use and there is good tool support available. The first step in the analysis is to define the system context, i.e. the overall sources and sinks for the data flows. For this case study, the monitored state variables provide the sources, and the controlled state variables provide the sinks. The system is then partitioned into its main component processes, and these are repeatedly partitioned into smaller-and-smaller subprocesses. Each partitioning is carried out in such a way that the data entering and leaving a parent process is preserved by its descendants. The results of the analysis are shown in Appendix A.

2.4.1 Context

Defining the system context presented some problems. In particular, not all the monitored variables actually appear in the system context; conversely, not all variables which do appear are monitored variables. For example, the WLMS water level does not appear in the system context, whereas differential pressure does. This is caused by the decision not to include tolerances in the VDM model.

2.4.2 Data Structures

The process of partitioning a system into subprocesses involves matching the flow of data at one level of decomposition with the next level of decomposition. This can be done in a fairly straightforward way by using composite data flows for the more abstract levels of decomposition, and singleton data flows for the more decomposed levels. This structuring technique can be carried across into VDM by using VDM record types for the composite data flows and simple types for the singleton data flows.

2.4.3 Base Level Processes

At some point in the partitioning process, we will have a number of processes which are manageable. At this point the partitioning is complete and a VDM specification can be constructed. The individual processes will then correspond to individual VDM functions or procedures. A complete listing of the VDM specification is given in Appendix B.
3. ANIMATION

One of the benefits of using VDM rather than natural language is that the VDM specification can be readily animated. This, in turn, means that the specification can be tested by someone with little or no understanding of the original VDM. To ensure that animation is as faithful as possible to the original VDM, the translation needs to be carried out as mechanically as possible. For this reason, we have avoided building an extensive graphical user interface, and we have tried to keep the translation rules as simple as possible. In this section, we describe the rules that were used to produce the animation from its VDM specification. The Pascal source code of the animation is given in Appendix C.

3.1 Transformation Rules

A large part of the VDM specification is given implicitly in terms of pre and post conditions. This has the advantage of leaving undefined the order of evaluation of any output variables. On the other hand, an animation must run sequentially, so a particular order of evaluation must be chosen. The translation strategy adopted was first of all to normalise each post condition according to the following re-write rules:

1. \( el \land (e2 \lor e3) = (el \land e2) \lor (el \land e3) \)
2. \( \neg (el \land e2) = (\neg el) \lor (\neg e2) \)
3. \( \neg (el \lor e2) = (\neg el) \land (\neg e2) \)
4. \( (el \Rightarrow e2) = (\neg el) \lor (el \land e2) \)

where \( el, e2, \) and \( e3 \) are boolean expressions.

This process results in normalised expressions of the form \((d1 \lor d2 \lor \ldots \lor dn)\), where each disjunction, \(d1\), is itself a conjunction of basic term, and is of the form \((ti1 \land ti2 \land \ldots)\). The next step is to identify those basic terms which set the values of any exported variables. These basic term are modelled in Pascal by assignment statements. The remaining terms define the conditions under which each assignment should take place. This can be modelled in Pascal by an IF-statement. Suppose for example that the basic term \(tij\) defines the value of the exported variable \(vi\). This can be modelled using a Pascal statement of the form \(vi := eij\). Taking account of the conditions under which the assignment can take place leads to the following code:

\[
\text{IF} \ (\text{ti1} \land \ldots \land \text{tij-1} \land \text{tij+1} \land \ldots) \\
\text{THEN } vi := eij
\]

This defines a translation strategy for a single disjunction. To cater for the general case we need to choose a particular order of evaluation. If we let \(iFi\) represent the above IF-statement then we can define a translation strategy for several disjunctions, \((d1 \lor d2 \lor \ldots \lor dn)\), as follows:

\[
\text{IF1 ELSE IF2 ELSE \ldots ELSE IFn}
\]
The above strategy provides us with a translation for each post condition. This can be combined with each pre condition as follows

\[ \text{IF pre condition THEN post condition} \]

In certain circumstances, an exported variable may be undefined. This can arise, for instance, when the pre condition is false or if any of the terms \( F_i \) above contains an empty assignment statement. This could be due to a specification error or it could be intentional. In either case, this situation can be catered for by initialising all exported variables to some undefined value. This can be done quite easily for variables of enumerated type by setting them initially to the enumerated value "undefined". For a variable of type real, this could be done by initialising it to a very large value.

3.1.1 Input-Output

The VDM specification uses sequences to model input and output. This was animated by reading and writing from text files, (see Appendix D). This approach led to an animation with minimal changes to the VDM specification. A more sophisticated animation e.g. one which included a real-time display, would be significantly different from the original VDM specification, and would be more likely to contain errors.

3.1.2 Make Functions

Make functions are used extensively throughout the specification. Each make function is associated with a particular record type and its name is formed by prefixing "mk-" to the name of the record type. When applied to appropriate values for the fields, a make function yields the value of the corresponding record type value. Using a make function to extract the value of each field (i.e. unpacking) was animated by careful use of the Pascal WITH-statement. Packing the fields was animated using special Pascal functions designed for this purpose.

3.1.3 Types

Most of the VDM data types used in the specification were modelled in a straightforward way using the corresponding Pascal data types. However, there is no direct Pascal equivalent of the VDM map type. This type was translated into a two dimensional Pascal array type.

3.1.4 Quantifiers

The VDM specification uses both existential and universal quantifiers in expressions. It is difficult to formulate practical translation rules for such expressions. One approach could be to replace each universally quantified expression by a series of conjunctions, and each existentially quantified expression by a series of disjunctions. This could be practical for quantification over enumerated types, but it would be impractical for quantification over large sets, e.g. the set of all integers. An alternative approach would be to use a loop statement. The
approach actually adopted was to transform any expression with a universal quantifier into a Pascal \texttt{WHILE-DO} loop. Translations for expressions containing existential quantifiers were carried out on an ad hoc basis.

4. \textbf{VERIFICATION}

In this section we consider two techniques for verifying a VDM specification against a user's requirements, and ensuring that it does not contain contradictions:

(1) Animation Testing
(2) Proving Theorems

In animation testing, test cases are constructed to simulate the expected behaviour of the system. The main deficiency of this approach is that it is normally impractical to simulate all possible system behaviour; any unusual system modes tend to be overlooked. An alternative approach to animation testing is to prove theorems about the system's behaviour. This is particularly useful for detecting any pathological behaviour.

4.1 \textbf{Animation Testing}

The VDM animation and, by implication, the VDM specification were tested by running a number of different transients. These were chosen to check different aspects of the WLMS's expected behaviour. Ideally, this task should be carried out by someone other than the author of the VDM specification, e.g. the originator of the requirements document. In this section we outline the tests that were conducted together with the test results.

4.1.1 \textbf{Operating Mode Transitions}

Once it has been initialised, the WLMS is in exactly one of the following operating modes: operating, standby, shutdown or test. The transitions from one mode to the next are defined by various triggered events. Test cases were generated by examining the conditions under which a triggered event can occur. The following test cases were selected by this means:

(1) Verify that the system is initially in standby mode.

(2) Verify that when the system is in standby mode, and the reset button has been pressed for more than 3s, and the level is within hysteresis range, then the system changes to operating mode.

(3) Verify that when the system is in either operating mode or standby mode or shutdown mode, and the selftest button has been pressed for more than 0.5s, then the system changes to test mode.

(4) Verify that when the system is in operating mode, and the water level drifts out of range, and the selftest button is not being pressed, then the system changes to shutdown mode.
(5) Verify that when the system is in shutdown mode, and the water level becomes within range in less than 0.2s, and the selftest button is not being pressed, then the system changes to operating mode.

(6) Verify that when the system is in shutdown mode, and the water level remains out of range for 0.2s, and the selftest button is not being pressed, then the system changes to standby mode.

(7) Verify that after the system has spent 14s in test mode it changes to standby mode.

4.1.2 Failure Mode Transitions

Once it has been initialised, the WLMS is in exactly one of the following failure modes: allok, badlevdev or hardfail. The following test cases were generated to test the transitions from one failure mode to the next:

(1) Verify that, initially, the system is in allok mode.

(2) Verify that when the system is in allok mode, and the level device fails while both control unit and the time device are not failed then the system mode changes to badlevdev.

(3) Verify that when the system is in either allok mode or badlevdev mode, and either the control unit fails or the time device fails, then the system mode changes to hardfail.

4.1.3 High(Low) Window

Test cases were generated to demonstrate the following expected behaviour:

(1) Demonstrate that both the high and low window are initially set to "on".

(2) Demonstrate that when the system enters operating mode and allok mode, and the water level is not high(low), then the high(low) window is set to "off"

(3) Demonstrate that when the system is in operating mode and allok mode, and the level drifts high(low), then the high(low), window is set to "on".

(4) Demonstrate that when the system is in shutdown mode and allok mode, and the level drifts high(low), then the high(low), window is set to "on".
(5) Demonstrate that when the system is in test mode and allok mode:
   (a) the high window is set to "on" for the first 2s and then set to "off".
   (b) the low window is set to "on" after 2s and then set to "off" after 4s.

(6) Demonstrate that when the system is in badlevdev mode the high and low windows flash alternatively "on" and "off" every 0.5s.

4.1.4 Alarm

Test cases were generated to demonstrate the following expected behaviour:

(1) Demonstrate that the alarm is set initially to "silent".

(2) Demonstrate that when the system enters operating mode and allok mode, and the water level is in range, then the alarm is set to "silent".

(3) Demonstrate that when the system is in operating mode and allok mode, and the level is out of range, then the alarm is set to "sound".

(4) Demonstrate that when the system is in shutdown mode and allok mode, and the level is out of range, then the alarm is set to "sound".

(5) Demonstrate that when the system is in test mode and allok mode, the alarm is set to "sound" for the first 4s and then set to "silent".

(6) Demonstrate that when the system enters badlevdev mode then the alarm is set to "sound".

4.1.5 Pump Switch

Test cases were generated to demonstrate the following expected behaviour:

(1) Demonstrate that when the power is "off" then the pump switch is open.

(2) Demonstrate that when the system enters operating mode and allok mode, and the reset button is released, then the pump switch is closed.

(3) Demonstrate that when the system enters any mode other than hardfail mode or operating mode or shutdown mode then the pump switch is opened.
4.1.6 Level Display

Test cases were generated to demonstrate the following expected behaviour:

(1) Demonstrate that when the system is in operating mode or shutdown mode or standby mode then display level changes according to the differential pressure as expected.

(2) Demonstrate that when the system is in test mode:
   (a) the level display is set to "0.0" for the first 4s
   (b) after 4s and for the next 9s the level display shows in succession
       "0.0", "1.1", "2.2", ..., "9.9"
   (c) after 13s the level display is set to "0.0"

4.1.7 Test Results

The animation was used to check the VDM specification for each of the above test cases. Most of the tests results were as expected. However, the tests uncovered a number of errors in both the VDM specification and in the reference [8.1]. The errors in the VDM specification were of 2 main types

(1) Omissions in the VDM specification, e.g. a missing case in a VDM post condition.

(2) Logical errors, e.g. the use of the VDM "and" function instead of an "or" function.

Reference [8.1] appears to contain an error in the definition of the level display for the case when the system is in test mode. The level display shows each of the values "0.0", "1.1", ..., "8.8" but it does not display "9.9".

4.2 Theorem Proving

A direct way of testing a VDM specification is to prove theorems about it. There are various sorts of theorems that could be proved. For example it would be possible to show there are no situations in which a post-condition could be false, i.e. for each operation/function we could attempt to prove that

\[
\forall \text{inp} : \text{inputs}. \exists \text{out} : \text{outputs} \quad \text{pre-condition(inp)} \Rightarrow \text{post-condition(inp, out)}
\]
where forall is the universal quantifier
there exists is the existential quantifier
"inp" represents the inports to the operation/function and
"out" represents the exports.

If, for example, we attempt to prove this theorem for the VDM function
WasinStandby in Appendix B then we will fail. This is because it is
possible for the variable "mode" to be set to "operating" and "test"
simultaneously. This can happen in the unlikely event that the reset
button is pressed exactly 2.5s before pressing the self test button. One
solution to this problem would be to tighten the pre-condition and so
evacuate this possibility. However, the resulting VDM would not be in
accordance with reference [8.1]. Alternatively, we could change the post-
condition to ensure that, in this non-deterministically.

Another class of theorems that could be proved applies to incomplete
specifications. It would be possible to prove that every exported
variable is defined, i.e.

\[
\text{forall } \text{inp} : \text{inputs. forall } \text{out} : \text{outputs}
\text{post condition}(\text{inp, out}) \implies (\text{out} \neq 1)
\]

where "inp" represents the inports to the operation/function,
"out" represents the exports and
1 represents the undefined value.

5. FUTURE WORK

The present work could be followed up in a number of ways. In this
section we make a number of suggestions which need to be discussed further
with AECB.

One area of work which follows naturally from the present study is program
verification. The work could involve exploring the stages and issues
involved in verifying a program against a VDM specification. For this it
would be desirable to make use of the SPADE static analysis tools (see
reference [8.7]). Verification could be carried out either at the source
code level or at the assembler level. Amongst the issues that such an
investigation would consider would be:

(1) Real Numbers. It is very difficult to formally verify programs
    which use floating point packages.

(2) Loop Termination. It is possible to verify that a program contains
    no non-terminating loops.
One important area which could be pursued further is concurrency. Theoretical approaches to this problem are still the subject of research. However, there are a number of promising approaches which have been developed over the years, for example:

1. Petri nets (reference [8.4])
2. Temporal logic (reference [8.5])
3. CSP (reference [8.6])

The suitability or otherwise of these methods for defining requirements could be investigated. This could involve pilot case studies and a review of any tool support.

A third area of study could be to investigate further the use of theorem proving in specification testing.

A fourth area of study could be to investigate how a VDM design could be produced from a VDM model of requirements.

6. VDM AND OTHER METHODS

VDM is one of a number of formal methods which can be used to specify a real-time system. It is a state-based method which allows abstraction and animation at the specification stage, and step-wise refinement towards an implementation. It is ideally suited to the specification and development of conventional sequential programs. On the other hand, VDM has no concept of time and does not support concurrency. Z is a formal method (see reference [8.8]) which is very similar to VDM except that it supports a schema calculus. This allows specifications to be constructed in an object-oriented manner. A well-established formal method which supports concurrency and has a model of time is CSP (see reference [8.6]). Unlike VDM, this method can be used to prove liveness properties of a concurrent system or absence of deadlock. CSP can be used to specify sequential programs, but is more suited to concurrent programming languages, e.g. occam (see reference [8.9]). A related formal language is CCS which has been used for formally specifying hardware systems. The notation of events and Event tables used in reference [8.1] is not really a formal method (i.e. it does not support step-wise refinement), but it does constitute a specification language. The language has a well-defined syntax, but the authors are not aware of the existence of a formal semantics, so, unlike Z, VDM and CSP, the exact meaning of the language is open to debate. Unlike VDM which supports a wide range of types (e.g. enumeration, record, and function types), the Event table approach supports very few data types, and this tends to complicate specifications. On the other hand, the Event table approach has a model of time, whereas Z and VDM do not. This model is, however, very detailed, and, for some concurrent systems, it may be more appropriate to use CSP.
7. CONCLUSIONS

This report has demonstrated the feasibility of using VDM to model the requirements of a simple real-time system, and how a Yourdon-style analysis can be employed to structure the specification. The report has also shown how an executable model of the VDM model, an animation, can be developed, and how this can be used to increase confidence in the VDM specification. For example, the animation testing demonstrated that the VDM specification faithfully defines various properties of the system; it also uncovered some errors in both the VDM model and reference [8.1]. Further confidence in the VDM model could be gained by applying reasoning directly to the VDM specification. The report has outlined some techniques by which this can be done. In particular, it has outlined how theorem proving can be used to establish that a VDM model is consistent and, in a certain sense, complete.

A number of areas of work have been suggested which could be investigated in a follow-up to this study.

8. REFERENCES


[8.8] J. M. Spivey,
Understanding Z. A specification language and its formal semantics,

[8.9] INMOS Ltd.,
occam - Programming Manual,

[8.10] E. Yourdon,
Techniques of Program Structure and Design,
This appendix shows the results of a Yourdon-style structured analysis applied to the WLMS. The diagrams were produced manually with the help of a CASE tool. The analysis includes the context diagram and the data flow between processes for different levels of decomposition. The decomposition has been carried out in such a way that the overall data flowing into and out of a process is preserved. The technique used for carrying out the analysis is described in reference [8.10].

Figure 1 shows the context of the WLMS and its interfaces with the outside world.

Figure 2 shows the principal processes within the WLMS i.e. the processes Initialise, Monitor and Normal Operation.

Figure 3 shows the decomposition of the Normal Operation process into the primary functions it must perform.

Figure 4 is a further decomposition of the process Control State Variables such that no further decomposition is necessary in order to produce the VDM specification.
Figure 1: CONTEXT DIAGRAM
Figure 2: Decomposition of HLMS
Figure 3: Decomposition of Normal Operation
Figure 4: Decomposition of Control State Variables
VDM SPECIFICATION

This appendix contains the VDM specification. Each page of VDM text is supplemented with an English commentary.

CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>B3</td>
</tr>
<tr>
<td>types</td>
<td>B5</td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>2.0.1 UpdateMonitoredInputs</td>
<td>B9</td>
</tr>
<tr>
<td>2.1 MonitorDevices</td>
<td>B11</td>
</tr>
<tr>
<td>2.2 UpdateCRT</td>
<td>B13</td>
</tr>
<tr>
<td>2.2.1 GetWindowChar</td>
<td>B11</td>
</tr>
<tr>
<td>2.2.2 ScreenBackground</td>
<td>B11</td>
</tr>
<tr>
<td>2.3 ControlStateVariables</td>
<td>B45</td>
</tr>
<tr>
<td>2.3.1 PumpControl</td>
<td>B15</td>
</tr>
<tr>
<td>2.3.2 GetDisplayData</td>
<td>B23</td>
</tr>
<tr>
<td>2.3.2.1 FlashOff</td>
<td>B15</td>
</tr>
<tr>
<td>2.3.2.2 GetHighWindow</td>
<td>B17</td>
</tr>
<tr>
<td>2.3.2.3 GetLowWindow</td>
<td>B19</td>
</tr>
<tr>
<td>2.3.2.4 GetLevelDisplay</td>
<td>B21</td>
</tr>
<tr>
<td>2.3.3 AlarmControl</td>
<td>B25</td>
</tr>
<tr>
<td>2.3.4 PumpEnvironment</td>
<td>B27</td>
</tr>
<tr>
<td>2.3.5 UpdateStoredData</td>
<td>B41</td>
</tr>
<tr>
<td>2.3.5.1 GetNextMode</td>
<td>B35</td>
</tr>
<tr>
<td>2.3.5.1.1 WasOperating</td>
<td>B27</td>
</tr>
<tr>
<td>2.3.5.1.2 WasShutdown</td>
<td>B29</td>
</tr>
<tr>
<td>2.3.5.1.3 WasinStandby</td>
<td>B31</td>
</tr>
<tr>
<td>2.3.5.1.4 WasinTest</td>
<td>B31</td>
</tr>
<tr>
<td>2.3.5.1.5 WasAlloc</td>
<td>B33</td>
</tr>
<tr>
<td>2.3.5.1.6 WasBadLevdev</td>
<td>B33</td>
</tr>
<tr>
<td>2.3.5.2 CheckButtonTimes</td>
<td>B37</td>
</tr>
<tr>
<td>2.3.5.3 GetInmodeTime</td>
<td>B37</td>
</tr>
</tbody>
</table>
2.3.5.4 CheckControlUnit..................... B39
2.3.5.5 CheckLevelDevice..................... B39
2.3.6 GetWaterLevel.......................... B43
2.3.7 CheckTimer.............................. B43
2.4 SendControlSignals....................... B47

operations

1.0 Initialise............................... B47
2.0 NormalOperation......................... B51
3.0 WLMS....................................... B51

Module Util..................................... B53
module WLMSSYSTEM

imports

from UTIL :
  functions
    MaxIntBelow : R |→ Z,
    Limit : R × R × R |→ R,
    round : R × Z |→ R,
   InRange : R × R × R |→ B,
    GetDigitChar : Z × R |→ Char

definitions

values

  spacechar : Char = " ";
  fillchar : Char is not yet defined;
  levelLowerCal : R = 13.0;
  levelUpperCal : R = 27.0;
  DPscale : R = 1.03803 × (levelUpperCal - levelLowerCal);
  DPoffset : R = -0.01902 × (levelUpperCal - levelLowerCal) +
    levelLowerCal;
  shutdownLockTime : R = 200;
  watchdogtimeout : R = 500;
  waterTolerance : R = 0.5;
  hysteresis : R = 0.5;
  highWaterLimit : R = 26.0;
  initTime : R is not yet defined;-- see precondition for Initialise */
  lowWaterLimit : R = 14.0;
  LDsclale : R = 11.1;
  maxAlarmTime : R = 4000;
  maxSelftestDelay : R = 500;
  maxResetDelay : R = 3000;
  minHWtestTime : R = 0;
  maxHWtestTime : R = 2000;
WLMSSYSTEM

Module WLMSSYSTEM imports the following functions from the VDM module UTIL:

MaxIntBelow - calculates the largest integer less than or equal to a given real number
Limit - limits a real number between a minimum and maximum value
Round - rounds a real number to the nearest integer
InRange - checks to see that a given number is in a given range
GetDigitChar - extracts a particular digit in a real number and returns the corresponding character

A number of constants (i.e. values) are defined.

spacechar  Space character
fillchar    Fill character i.e. reverse of a space
levelLowerCal  Lower calibration bound (cm)
levelUpperCal  Upper calibration bound (cm)
DPscale  Scale factor on differential pressure
DPOffset  Differential pressure offset
shutdownLockTime  Shutdown lock time (ms)
watchdogtimeout  Watchdog timeout (ms)
waterTolerance  Water tolerance (cm)
hysteresis  Hysteresis (cm)
highWaterLimit  High water limit (cm)
initTime  Satisfies 0 <= initTime <= 5
lowWaterLimit  Low water limit (cm)
LDscale  Level display scale factor
maxAlarmTime  Maximum time in *test* mode
maxSelftestDelay  Minimum time the selftest button needs to be pressed (ms)
maxResetDelay  Minimum time the reset button needs to be pressed (ms)
minHWtestTime  Time in *test* mode at which the high window goes "on"
maxHWtestTime  Time in *test* mode at which the high window goes "off"
\begin{verbatim}
minLWtestTime : \mathbb{R} = 2000;
maxLWtestTime : \mathbb{R} = 4000;
minLDTime : \mathbb{R} = 4000;
maxLDTime : \mathbb{R} = 13000;
maxTestDelay : \mathbb{R} = 14000

types
  byte = \mathbb{Z}
  \quad \text{inv}(z) \Delta z \in \{0, \ldots, 255\};

deviceType = \{ok, failed\};
watchDogType = \{uninit, operate, shut\};
onOffType = \{on, off\};
buttonType = \{pressed, released\};
operatingModeType = \{operating, shutdown, standby, test\};
failureModeType = \{allok, badlevdev, hardfail\};
rows = \mathbb{Z}
  \quad \text{inv}(p) \Delta p \in \{1, \ldots, 25\};

columns = \mathbb{Z}
  \quad \text{inv}(p) \Delta p \in \{1, \ldots, 40\};

gridpoint = rows \times columns;
screenType = gridpoint \xrightarrow{m} \text{Char};
pumpSwitchType = \{open, closed\};
shutdownSignalType = \{operate, shutdown\};
alarmType = \{silent, audible\};

buttonTimesType ::
  resetButtonTime : \mathbb{R}
  selftestButtonTime : \mathbb{R};

modesType ::
  operatingMode : operatingModeType
  failureMode : failureModeType;

storedVarType ::
  highWindow : onOffType
  lowWindow : onOffType
  alarm : alarmType
  shutdownSignal : shutdownSignalType;
\end{verbatim}
This section defines the types used in the remainder of the module

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minLWtestTime</td>
<td>Time in <em>test</em> mode at which the low window goes &quot;on&quot;</td>
</tr>
<tr>
<td>maxLWtestTime</td>
<td>Time in <em>test</em> mode at which the low window goes &quot;off&quot;</td>
</tr>
<tr>
<td>minLDTtime</td>
<td>End of the first stage in <em>test</em> mode</td>
</tr>
<tr>
<td>maxLDTtime</td>
<td>End of the second stage in <em>test</em> mode</td>
</tr>
<tr>
<td>maxTestDelay</td>
<td>Maximum time spent in <em>test</em> mode</td>
</tr>
</tbody>
</table>
storedDataType ::
  time : R
  timeInMode : R
  watchDogTime : R
  modes : modesType
  buttonTimes : buttonTimesType;

controlSignalsType ::
  alarm : alarmType
  shutdownSignal : shutdownSignalType
  pumpSwitch : pumpSwitchType
  watchdog : watchDogType;

monitorVarType ::
  diffPress : byte
  resetButton : buttonType
  selftestButton : buttonType
  powerNow : onOffType
  memory : deviceType
  timeNow : R
  timeDevice : deviceType;

crtDataType ::
  highWindow : onOffType
  lowWindow : onOffType
  levelDisplay : R;

crtdVarType ::
  crtdata : crtDataType
  controlSignals : controlSignalsType
  storeNow : storedDataType
  variablesNow : storedVarType;

monitorInputType ::
  diffPressList : byte*
  resets : buttonType*
  selftests : buttonType*
  powerSignal : onOffType*
  memorySeq : deviceType*
  timeSeq : R*
  timeDeviceSeq : deviceType*;

controlOutputType ::
  shutdownSignals : shutdownSignalType*
  watchDogSignals : watchDogType*
  pumpSignals : pumpSwitchType*
  alarmSignals : alarmType*;

systemOutputType ::
  controlOutputs : controlOutputType
  displaySignals : screenType*
This section defines the types used in the remainder of the module.
state \textit{WLMS} of

\begin{itemize}
  \item storedData : \textit{storedDataType}
  \item storedVariables : \textit{storedVarType}
  \item monitoredInputs : \textit{monitorInputType}
  \item systemOutput : \textit{systemOutputType}
\end{itemize}

end

\section*{FUNCTIONS}

functions

\section*{— 2.0.1 Update Monitored Inputs}

\begin{verbatim}
UpdateMonitoredInputs(initial : monitorInputType)
  final : monitorInputType
pre true
post let mk-monitorInputType(dp0, rst0, slf0, pwr0, mem0, tim0, dev0) = initial in
  let mk-monitorInputType(dp, rst, slf, pwr, mem, tim, dev) = final in
  (dp = tl dp0) \land (rst = tl rst0) \land (slf = tl slf0) \land 
  (pwr = tl pwr0) \land (mem = tl mem0) \land (tim = tl tim0) \land 
  (dev = tl dev0);
\end{verbatim}
The state consists of the following record type variables:

- **storedData**
  - Data store containing:
    - (1) current time
    - (2) time in the current mode
    - (3) time since the watchdog timer was reset
    - (4) current modes
    - (5) time the reset button and self test button have been pressed

- **storedVariables**
  - Controlled-variables store containing:
    - (1) current state of the high window
    - (2) current state of the low window
    - (3) current state of the alarm
    - (4) current value of the shutdown signal

- **monitoredInputs**
  - Monitored variables containing:
    - (1) differential pressure cell
    - (2) reset button
    - (3) self test button
    - (4) system power supply
    - (5) state of the computer's memory
    - (6) system times
    - (7) state of the time device

- **systemOutputs**
  - Controlled variables containing:
    - (1) shutdown signals
    - (2) watch dog signals
    - (3) pump signals
    - (4) alarm signals
    - (5) CRT display

The quantitative interpretations of these variables are given by the types and invariants defined on pages B5-B8.

2.0.1 UpdateMonitoredInputs

This function updates the monitored inputs. This is carried out by first extracting the components of "initial" i.e.

- dp0 Sequence of differential pressure values
- rst0 Sequence of reset button values
- slf0 Sequence of selftest button values
- mem0 Sequence of memory states
- tim0 Sequence of times
- dev0 Sequence of device states
- pwr0 Sequence of power values

Each of these components is updated by extracting the tail from each sequence of values.
2.1 Monitor Devices Function

\[ \text{MonitorDevices} \left( \text{inputs} : \text{monitorInputType} \right) \text{monitored} : \text{monitorVarType} \]

pre true
post let \( \text{mk-monitorInputType}(dpcell, \text{resets, selftests, powers, memory, times, timedev}) = \text{inputs} \) in

\[ \text{monitored} = \text{mk-monitorVarType}(hd \ dpcell, \text{resets, selftests, powers, memory, times, timedev}); \]

2.2.1 Get Window Character Function

\[ \text{GetWindowChar} \left( \text{window} : \text{onOffType} \right) \ c : \text{Char} \]

pre true
post ( (\text{window} = \text{off}) \land (c = \text{spacechar}) \\
\lor (\text{window} = \text{on}) \land (c = \text{fillchar}) \\
);

2.2.2 Screen Background Function

\[ \text{ScreenBackground} () \ \text{background} : \text{screenType} \]

pre true
post true; -- i.e. is not yet defined
2.1 MonitorDevices

This function "reads" the current value of the monitored inputs. Each component is "read" using the unary function $H_d$ which extracts the head of a sequence.

2.2.1 GetWindowChar

This function returns a space character if window = "off", and a fill character if window = "on".

2.2.2 ScreenBackground

This function left undefined. Any implementation will satisfy the post condition.
2.2 Update CRT Display Function

UpdateCRT\( (\text{initOutput : screenType}^*, \text{crtData : crtDataType}) \)
\text{finalOutput : screenType}^*

\text{pre true}

\text{post let mk-crtDataType(highWindow, lowWindow, levelDisplay) =}
\text{crtData in}
\text{let hwchar = GetWindowChar(highWindow) in}
\text{let lwchar = GetWindowChar(lowWindow) in}
\text{let digit1 = UTIL'GetDigitChar(1, levelDisplay) in}
\text{let digit2 = UTIL'GetDigitChar(2, levelDisplay) in}
\text{let digit3 = UTIL'GetDigitChar(3, levelDisplay) in}
\text{let digit4 = UTIL'GetDigitChar(4, levelDisplay) in}
\text{let screen = ScreenBackground \begin{array}{c}
(18,10) \rightarrow \text{hwchar}, \\
(18,30) \rightarrow \text{lwchar}, \\
(6,20) \rightarrow \text{digit1}, \\
(6,21) \rightarrow \text{digit2}, \\
(6,22) \rightarrow \text{digit3}, \\
(6,24) \rightarrow \text{digit4}
\end{array} \text{ in}
\text{finalOutput = initOutput \sim [screen];}
2.2 UpdateCRT

This function updates the CRT display. Screen is formed by superimposing the high window, low window and level display on the screen background. The output sequence, `finalOutput`, is obtained by appending the singleton sequence, `[screen]`, to the sequence `initOutput`. 
2.3.1 Pump Control Function

\[PumpControl(\text{oldSignal} : \text{shutdownSignalType},\]
\[\text{resetButton} : \text{buttonType}, \text{modes} : \text{modesType})\]
\[\text{shutdownSignal} : \text{shutdownSignalType}\]

\[\pre \text{true}\]
\[\post \text{let mk-modesType(opmode, failmode) = modes in}\]
\[
((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allok}) \land
\neg(\text{resetButton} = \text{pressed}) \Rightarrow (\text{shutdownSignal} = \text{operate}))
\land


\[((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allok}) \land
(\text{resetButton} = \text{pressed}) \Rightarrow (\text{shutdownSignal} = \text{oldSignal}))
\land


\[((\text{opmode} = \text{shutdown}) \land (\text{failmode} = \text{allok}) \Rightarrow
(\text{shutdownSignal} = \text{oldSignal}))
\land


\[((\text{opmode} \in \{\text{standby, test}\}) \land (\text{failmode} = \text{allok}) \Rightarrow
(\text{shutdownSignal} = \text{shutdown}))
\land


\[((\text{failmode} \in \{\text{badlevdev, hardfail}\}) \Rightarrow
(\text{shutdownSignal} = \text{shutdown});\]

--

2.3.2.1 FlashOff Function

\[\text{FlashOff(miliseconds : R) flash : B}\]

\[\pre \text{true}\]
\[\post \text{let residue = UTIL'MaxIntBelow(miliseconds/500) mod 2 in}\]
\[
\text{flash = (residue = 0);}\]
2.3.1 PumpControl

This function sets the value of the shutdown signal. It depends on the current value of the system operating mode and the system failure mode.

If the system operating mode is "operating", the system failure mode is "allok" and the reset button is not pressed then shutdown signal is set to "operate".

If the system operating mode is "operating", the system failure mode is "allok" and the reset button is pressed then shutdown signal is set to its previous value.

If the system operating mode is "shutdown" and the system failure mode is "allok" then shutdown signal is set to its previous value.

If the system operating mode is "standby" or "test", and the system failure mode is "allok" then the shutdown signal is set to "shutdown".

If the system failure mode is "badlevdev" or "hardfail" then the shutdown signal is set to "shutdown".

2.3.2.1 FlashOff

This function alternates between "true" and "false" every 500ms.
2.3.2.2 Get High Window Function

\[
\text{GetHighWindow}(\text{oldHighWindow} : \text{onOffType}, \text{waterlevel} : \text{R}, \text{time} : \text{R}, \text{timeInMode} : \text{R}, \text{modes} : \text{modesType}, \text{powerNow} : \text{onOffType})
\]

\[
\text{highWindow} : \text{onOffType}
\]

pre let \(\text{mk-modesType}(\text{opmode}, \text{failmode}) = \text{modes}\) in

\[
(\text{powerNow} = \text{on}) \land \neg(\text{failmode} = \text{hardfail})
\]

post let \(\text{mk-modesType}(\text{opmode}, \text{failmode}) = \text{modes}\) in

let \(\text{ishigh} = \text{waterlevel} \geq \text{highWaterLimit}\) in

let \(\text{enter} = (\text{timeInMode} = 0)\) in

let \(\text{itestring} = (\text{minHWlestTime} < \text{timeInMode}) \land\)

\(\text{timeInMode} < \text{maxHWlestTime}\) in

\[
((\text{opmode} \in \{\text{operating, shutdown}\}) \land (\text{failmode} = \text{alkok}) \land
\text{ishigh} \Rightarrow (\text{highWindow} = \text{on}))
\]

\[
((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{alkok}) \land \text{enter} \land
\neg\text{ishigh} \Rightarrow (\text{highWindow} = \text{off}))
\]

\[
((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{alkok}) \land \neg\text{enter} \land
\neg\text{ishigh} \Rightarrow (\text{highWindow} = \text{oldHighWindow}))
\]

\[
((\text{opmode} = \text{shutdown}) \land (\text{failmode} = \text{alkok}) \land
\neg\text{ishigh} \Rightarrow (\text{highWindow} = \text{oldHighWindow}))
\]

\[
((\text{opmode} = \text{standby}) \land (\text{failmode} = \text{alkok})) \Rightarrow
(\text{highWindow} = \text{oldHighWindow})
\]

\[
((\text{opmode} = \text{test}) \land (\text{failmode} = \text{alkok}) \land
\text{itestring} \Rightarrow (\text{highWindow} = \text{on}))
\]

\[
((\text{opmode} = \text{test}) \land (\text{failmode} = \text{alkok}) \land
\neg\text{itestring} \Rightarrow (\text{highWindow} = \text{off}))
\]

\[
((\text{failmode} = \text{badlevdev}) \land \text{FlashOff}(\text{time}) \Rightarrow (\text{highWindow} = \text{on}))
\]

\[
((\text{failmode} = \text{badlevdev}) \land \neg\text{FlashOff}(\text{time}) \Rightarrow (\text{highWindow} = \text{off}));
\]
2.3.2.2 GetHighWindow

This function sets the high window display. It depends on the previous setting, the water level, the time spent in the current mode, the current operating mode and the current failure mode. If the power is "off" or the failure mode is "hardfail" then the high window display is undefined.

If the system operating mode is "operating" or "shutdown", the system failure mode is "allok" and the water level is high then the display is "on".

When the system first enters "operating" mode, the display is "off" providing the water level is not high.

If the system is already in "operating" mode, the display is unchanged providing the water level is not high.

If the system operating mode is "shutdown", the system failure mode is "allok" and the water level is not high then the display is unchanged.

If the system operating mode is "standby" and the system failure mode is "allok" then the display is unchanged.

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is not greater than 2s then the display is "on".

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is greater than 2s then the display is "off".

While the system failure mode is "badlevdev" the display flashes "on" and "off".

GetLowWindow(\text{oldLowWindow} : \text{onOffType}, \text{waterlevel} : \mathbb{R}, \text{time} : \mathbb{R}, \text{timeInMode} : \mathbb{R}, \text{modes} : \text{modesType}, \text{powerNow} : \text{onOffType})

\text{lowWindow} : \text{onOffType}

\text{pre let mk-modesType(opmode, failmode) = modes in}

\quad (\text{powerNow} = \text{on}) \land \neg (\text{failmode} = \text{hardfail})

\text{post let mk-modesType(opmode, failmode) = modes in}

\quad \text{let islow} = \text{waterlevel} \leq \text{lowWaterLimit} \text{ in}

\quad \text{let enter} = (\text{timeInMode} = 0) \text{ in}

\quad \text{let istesting} = (\text{minLWtestTime} \leq \text{timeInMode}) \land

\quad \quad (\text{timeInMode} < \text{maxLWtestTime}) \text{ in}

\quad \quad (((\text{opmode} \in \{\text{operating, shutdown}\}) \land

\quad \quad \quad (\text{failmode} = \text{allow}) \land \text{islow}) \Rightarrow \text{(lowWindow = on)})

\quad \quad \land

\quad \quad (((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allow}) \land \neg \text{islow}) \Rightarrow \text{(lowWindow = off)})

\quad \quad \land

\quad \quad (((\text{opmode} = \text{operating}) \land (\text{failmode} = \text{allow}) \land \neg \text{enter} \land

\quad \quad \quad \neg \text{islow}) \Rightarrow \text{(lowWindow = oldLowWindow)})

\quad \quad \land

\quad (((\text{opmode} = \text{shutdown}) \land (\text{failmode} = \text{allow}) \land

\quad \quad \quad \neg \text{islow}) \Rightarrow \text{(lowWindow = oldLowWindow)})

\quad \land

\quad (((\text{opmode} = \text{standby}) \land (\text{failmode} = \text{allow})) \Rightarrow

\quad \quad \text{(lowWindow = oldLowWindow)})

\quad \land

\quad (((\text{opmode} = \text{test}) \land (\text{failmode} = \text{allow}) \land

\quad \quad \text{istesting}) \Rightarrow \text{(lowWindow = on)})

\quad \land

\quad (((\text{opmode} = \text{test}) \land (\text{failmode} = \text{allow}) \land

\quad \quad \neg \text{istesting}) \Rightarrow \text{(lowWindow = off)})

\quad \land

\quad ((\text{failmode} = \text{badlevdev}) \land \text{FlashOff(time) \Rightarrow (highWindow = off)})

\quad \land

\quad ((\text{failmode} = \text{badlevdev}) \land \neg \text{FlashOff(time) \Rightarrow (highWindow = on)};

\text{-B19-}
2.3.2.3 GetLowWindow

This function sets the low window display. It depends on the previous setting, the water level, the time spent in the current mode, the current operating mode and the current failure mode. If the power is "off" or the failure mode is "hardfail" then the low window display is undefined.

If the system operating mode is "operating" or "shutdown", the system failure mode is "allok" and the water level is low then the display is "on".

When the system first enters "operating" mode, the display is "off" providing the water level is not low.

If the system is already in "operating" mode, the display is unchanged providing the water level is not low.

If the system operating mode is "shutdown", the system failure mode is "allok" and the water level is not low then the display is unchanged.

If the system operating mode is "standby" and the system failure mode is "allok" then the display is unchanged.

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is between 2s and 4s then the display is "on".

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is not between 2s and 4s then the display is "off".

While the system failure mode is "badlevdev" the display flashes "on" and "off".
2.3.2.4 Get Level display Function

---

```plaintext
GetLevelDisplay(waterlevel: R, time: R,
               timeInMode: R, modes: modesType)
levelDisplay: R

pre let mk-modesType(opmode, failmode) = modes in
failmode = allok

post let mk-modesType(opmode, failmode) = modes in
let stage1 = (0 ≤ timeInMode) ∧
              (timeInMode < minLDTime) in
let stage2 = (minLDTime ≤ timeInMode) ∧
              (timeInMode < maxLDTime) in
let stage3 = (timeInMode ≥ maxLDTime) in

--- *** the display time must be scaled by 1000 !!!

let displaytime = (timeInMode - minLDTime)/1000 in

((opmode ∈ {operating, shutdown, standby}) ⇒
 (levelDisplay = UTIL'round(waterlevel,1)))
∧
(((opmode = test) ∧ stage1) ⇒ (levelDisplay = 0))
∧
(((opmode = test) ∧ stage2) ⇒
 (levelDisplay = LDscale × UTIL'MaxIntBelow(displaytime)))
∧
(((opmode = test) ∧ stage3) ⇒ (levelDisplay = 0));
```
2.3.2.4 GetLevelDisplay

This function sets the level display. It depends on the time spent in the current mode and the current operating mode. The function UTIL'MaxIntBelow is imported from module UTIL. If the failure mode is not "allok" then the level display is undefined.

If the system operating mode is "operating" or "shutdown" or "standby" then the level display is equal to the water level rounded to 1 decimal place.

If the system operating mode is "test" then the level display is either 0 or a multiple of 11.1.
-- 2.3.2 Get Display Data Function

GetDisplayData(\texttt{oldHighWindow} : \texttt{onOffType}, \texttt{oldLowWindow} : \texttt{onOffType}, \texttt{waterlevel} : \texttt{R}, \texttt{time} : \texttt{R}, \texttt{timeInMode} : \texttt{R}, \texttt{modes} : \texttt{modesType}, \texttt{powerNow} : \texttt{onOffType})
\texttt{crtData} : \texttt{crtDataType}

\textbf{pre} true
\textbf{post} let \texttt{mk-crtDataType(highWindow, lowWindow, levelDisplay)} =
\texttt{crtData} in

\texttt{highWindow} = \texttt{GetHighWindow(oldHighWindow, waterlevel, time, timeInMode, modes, powerNow)} \land

\texttt{lowWindow} = \texttt{GetLowWindow(oldLowWindow, waterlevel, time, timeInMode, modes, powerNow)} \land

\texttt{levelDisplay} = \texttt{GetLevelDisplay(waterlevel, time, timeInMode, modes)};
2.3.2 GetDisplayData

This function calculates the current high window display, low window display and level display.
2.3.3 Alarm Control Function

```plaintext
AlarmControl(oldAlarm : alarmType, waterlevel : R,
timeInMode : R, modes : modesType,
powerNow : onOffType)
alarm : alarmType

pre  let mk-modesType(opmode, failmode) = modes in
     (powerNow = on) \& \neg(failmode = hardfail)

post let mk-modesType(opmode, failmode) = modes in
     let isinlimits = UTILInRange(lowWaterLimit, waterlevel, highWaterLimit) in
     let isalarm = (0 \leq timeInMode) \& (timeInMode < maxAlarmTime) in
     let enter = (0 = timeInMode) in
     ((opmode \in \{shutdown, operating\}) \& (failmode = allok) \&
      \negisinlimits => (alarm = audible))
     \&
     ((opmode = operating) \& (failmode = allok) \& enter \&
      isinlimits => (alarm = silent))
     \&
     ((opmode = operating) \& (failmode = allok) \& \negenter \&
      isinlimits => (alarm = oldAlarm))
     \&
     ((opmode = shutdown) \& (failmode = allok) \& isinlimits =>
      (alarm = oldAlarm))
     \&
     ((opmode = standby) \& (failmode = allok) => (alarm = oldAlarm))
     \&
     ((opmode = test) \& (failmode = allok) \&
      isalarm => (alarm = audible))
     \&
     ((opmode = test) \& (failmode = allok) \&
      \negisalarm => (alarm = silent))
     \&
     ((failmode = badlevdev) \& enter => (alarm = audible))
     \&
     ((failmode = badlevdev) \& \negenter => (alarm = oldAlarm));
```
2.3.3 AlarmControl

This function sets the alarm. It depends on the previous setting, the water level, the time spent in the current mode, the current operating mode and the current failure mode. All cases have been considered. If the power is "off" or the failure mode is "hardfail" then the high alarm is undefined.

If the system operating mode is "operating" or "shutdown", the failure mode is "allok" and the water level is out of range then the alarm sounds.

When the system first enters "operating" and the water level is in range then the alarm is silent.

If the system is already in "operating" mode and the water level is in range then the state of the alarm is unchanged.

If the system operating mode is "shutdown" or "standby", the failure mode is "allok" and the water level is in range then the state of the alarm is unchanged.

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is not greater than 4s then the alarm sounds.

If the system operating mode is "test", the system failure mode is "allok" and the time in mode is greater than 4s then the alarm is silent.

When the system first enters "badlevdev" failure mode the alarm sounds.

If the system is already in "badlevdev" failure mode then the state of the alarm is unchanged.
2.3.4 Pump Environment Function

\[ \text{PumpEnvironment} \left( \text{powerNow} : \text{onOffType}, \text{shutdownSignal} : \text{shutdownSignalType}, \right. \]
\[ \left. \text{watchDog} : \text{watchDogType} \right) \]
\[ \text{pumpSwitch} : \text{pumpSwitchType} \]

\( \text{pre true} \)

\( \text{post } ((\text{powerNow} = \text{on}) \land (\text{shutdownSignal} = \text{operate}) \land (\text{watchDog} = \text{operate}) \)
\[ \Rightarrow (\text{pumpSwitch} = \text{closed}) \]
\[ \land \]
\( ((\text{powerNow} = \text{off}) \lor (\text{shutdownSignal} = \text{shutdown}) \lor (\text{watchDog} = \text{shut}) \)
\[ \Rightarrow (\text{pumpSwitch} = \text{open}) \];

2.3.5.1.1 Was in Operating Mode

\[ \text{WasOperating} \left( \text{initmode} : \text{operatingModeType}, \right. \]
\[ \left. \text{selftestButton} : \text{buttonType}, \text{waterlevel} : \mathbb{R}, \right. \]
\[ \left. \text{selftestButtonTime} : \mathbb{R} \right) \]
\[ \text{mode} : \text{operatingModeType} \]

\( \text{pre } \text{initmode} = \text{operating} \)

\( \text{post let } \text{inshut} = (\text{selftestButton} \neq \text{pressed}) \land \)
\[ \neg \text{UTIL'InRange}(\text{lowWaterLimit}, \text{waterlevel}, \]
\[ \text{highWaterLimit}) \) \in \]
\[ \text{let } \text{intest} = (\text{selftestButton} = \text{pressed}) \land \]
\[ (\text{selftestButtonTime} \geq \text{maxSelftestDelay}) \) \in \]
\[ (\text{inshut} \Rightarrow (\text{mode} = \text{shutdown})) \]
\[ \land \]
\[ (\text{intest} \Rightarrow (\text{mode} = \text{test})) \]
\[ \land \]
\[ (\neg \text{intest} \land \neg \text{inshut} \Rightarrow (\text{mode} = \text{operating}) \); \]
2.3.4 PumpEnvironment

This function sets the pump switch to "open" or "closed". It depends on the power, the shutdown signal and the watchdog signal.

2.3.5.1.1 WasOperating

This function defines the next operating mode when the system is, initially, in "operating".

If the selftest button is not pressed and the water level is out of range then the next mode is "shutdown"

If the selftest button is pressed for more than 500ms then the next mode is "test".

If none of these conditions hold then the mode is unchanged i.e. "operating".
2.3.5.1.2 Was in Shutdown Mode

\[\text{WasShutdown}(\text{initmode} : \text{operatingModeType}, \]
\[\quad \text{selftestButton} : \text{buttonType}, \text{waterlevel} : \mathbb{R}, \]
\[\quad \text{timeInMode} : \mathbb{R}, \text{selftestButtonTime} : \mathbb{R}) \]
\[\text{mode} : \text{operatingModeType} \]

**pre** \( \text{initmode} = \text{shutdown} \)

**post**
\[\text{let isinhyst} = \text{UTILInRange}(\text{lowWaterLimit} + \text{hysteresis}, \]
\[\quad \text{waterlevel}, \text{highWaterLimit} - \text{hysteresis}) \]
\[\text{let inop1} = (\text{selftestButton} \neq \text{pressed}) \land \]
\[\quad (\text{timeInMode} < \text{shutdownLockTime}) \land \text{isinhyst} \]
\[\text{let instnd1} = (\text{selftestButton} \neq \text{pressed}) \land \]
\[\quad (\text{timeInMode} \geq \text{shutdownLockTime}) \]
\[\text{let intest} = (\text{selftestButton} = \text{pressed}) \land \]
\[\quad (\text{selftestButtonTime} \geq \text{maxSelftestDelay}) \]

\[\quad (\text{inop1} \Rightarrow (\text{mode} = \text{operating})) \]
\[\land \]
\[\quad (\text{instnd1} \Rightarrow (\text{mode} = \text{standby})) \]
\[\land \]
\[\quad (\text{intest} \Rightarrow (\text{mode} = \text{test})) \]
\[\land \]
\[\quad (\neg \text{inop1} \land \neg \text{instnd1} \land \neg \text{intest} \Rightarrow (\text{mode} = \text{shutdown})) \]
2.3.5.1.2 WasShutdown

This function defines the next operating mode when the system is, initially, in "shutdown".

If the selftest button is not pressed, the water level is inside the hysteresis window, and the time in mode is less than the shutdown lock time then the next mode is "operating".

If the selftest button is not pressed, and the time in mode is greater than the shutdown lock time then the next mode is "standby".

If the selftest button is pressed for more than 500ms then the next mode is "test";

If none of these conditions hold then the mode is unchanged i.e. "shutdown".
-- 2.3.5.1.3 Was in Standby Mode

\[
\text{WasInStandby}(\text{initmode} : \text{operatingModeType}, \\
\text{selftestButton} : \text{buttonType}, \text{waterlevel} : \text{R}, \\
\text{resetButtonTime} : \text{R}, \text{selftestButtonTime} : \text{R}) \\
\text{mode} : \text{operatingModeType}
\]

\[
\text{pre initmode} = \text{standby} \\
\text{post let isinhyst} = \text{UTILInRange}(\text{lowWaterLimit}+\text{hysteresis}, \\
\text{waterlevel}, \text{highWaterLimit}−\text{hysteresis}) \text{ in} \\
\text{let inop2} = (\text{resetButtonTime} ≥ \text{maxResetDelay}) \land \text{isinhyst} \text{ in} \\
\text{let intest} = (\text{selftestButton} = \text{pressed}) \land \\
(\text{selftestButtonTime} ≥ \text{maxSelftestDelay}) \text{ in} \\
(\text{inop2} \Rightarrow (\text{mode} = \text{operating})) \\
\land \\
(\text{intest} \Rightarrow (\text{mode} = \text{test})) \\
\land \\
(\neg \text{inop2} \land \neg \text{intest}) \Rightarrow (\text{mode} = \text{standby})
\]

-- 2.3.5.1.4 Was in Test Mode

\[
\text{WasInTest}(\text{initmode} : \text{operatingModeType}, \text{timeInMode} : \text{R}) \\
\text{mode} : \text{operatingModeType}
\]

\[
\text{pre initmode} = \text{test} \\
\text{post let instnd2} = (\text{timeInMode} ≥ \text{maxTestDelay}) \text{ in} \\
(\text{instnd2} \Rightarrow (\text{mode} = \text{standby})) \\
\land \\
(\neg \text{instnd2}) \Rightarrow (\text{mode} = \text{test})
\]
2.3.5.1.3 WasinStandby

This function defines the next operating mode when the system is, initially, in "standby".

If the reset button is pressed for more than 3s and the water level is inside the hysteresis window then the next mode is "operating".

If the selftest button is pressed for more than 500ms then the next mode is "test";

If none of these conditions hold then the mode is unchanged i.e. "standby".

2.3.5.1.4 WasinTest

This function defines the next operating mode when the system is, initially, in "test".

If the time in this mode exceeds 14s then the next mode is "standby".

If none of these conditions hold then the mode is unchanged i.e. "test".
-- 2.3.5.1.5 Was in Allok Mode

\[
\text{WasAllok}(\text{initmode} : \text{failureModeType}, \\
    \text{controlUnit} : \text{deviceType}, \text{levelDevice} : \text{deviceType}, \\
    \text{timeDevice} : \text{deviceType}) \\
\text{mode} : \text{failureModeType}
\]

pre \( \text{initmode} = \text{allok} \)
post let \( \text{hrd} = (\text{controlUnit} = \text{failed}) \lor (\text{timeDevice} = \text{failed}) \) in
let \( \text{lev} = (\text{levelDevice} = \text{failed}) \land (\text{controlUnit} = \text{ok}) \land \\
(\text{timeDevice} = \text{ok}) \) in
\[
(\text{lev} \Rightarrow (\text{mode} = \text{badlevdev})) \land \\
(\text{hrd} \Rightarrow (\text{mode} = \text{hardfail})) \land \\
((\neg \text{lev} \land \neg \text{hrd}) \Rightarrow (\text{mode} = \text{allok}))
\]

-- 2.3.5.1.6 Was in BadLevdev Mode

\[
\text{WasBadLevdev}(\text{initmode} : \text{failureModeType}, \\
    \text{controlUnit} : \text{deviceType}, \text{levelDevice} : \text{deviceType}, \\
    \text{timeDevice} : \text{deviceType}) \\
\text{mode} : \text{failureModeType}
\]

pre \( \text{initmode} = \text{badlevdev} \)
post let \( \text{hrd} = (\text{controlUnit} = \text{failed}) \lor (\text{timeDevice} = \text{failed}) \) in
\[
(\text{hrd} \Rightarrow (\text{mode} = \text{hardfail})) \land \\
(\neg \text{hrd} \Rightarrow (\text{mode} = \text{badlevdev}))
\]
2.3.5.1.5 WasAllok

This function defines the next failure mode when the system is, initially, in "allok".

If the control unit and the time device are operating, but the level device has failed then set the next mode to "badlevdev".

If either the control unit or the time device has failed then set the next mode to "hardfail".

If none of these conditions hold then the mode is unchanged i.e. "allok".

2.3.5.1.6 WasBadlevdev

This function defines the next failure mode when the system is, initially, in "badlevdev".

If either the control unit or the time device has failed then set the next mode to "hardfail".

If neither of these conditions hold then the mode is unchanged i.e. "badlevdev".
2.3.5.1 Get Next Mode

\[
\text{GetNextMode}(\text{waterlevel} : \mathbb{R}, \text{resetButton} : \text{buttonType}, \\
\text{selftestButton} : \text{buttonType}, \text{store} : \text{storedDataType}, \\
\text{controlUnit} : \text{deviceType}, \text{levelDevice} : \text{deviceType}, \\
\text{timeDevice} : \text{deviceType}) \\
\rightarrow \text{newmodes} : \text{modesType}
\]

pre true

post let mk-storedDataType(time, timeInMode, watchdogTime, \\
\text{modes}, \text{buttonTimes}) = \text{store} in
let mk-modesType(opmode, failmode) = \text{modes} in
let mk-modesType(newopmode, newfailmode) = \text{newmodes} in
let mk-buttonTimesType(resetButtonTime, selftestButtonTime) = \\
\text{buttonTimes} in

\[
\begin{align*}
((\text{opmode} = \text{operating}) \land \\
(\text{newopmode} = \text{WasOperating}(\text{opmode}, \text{selftestButton}, \text{waterlevel}, \\
\text{selftestButtonTime}) ) \\
\lor \\
(\text{opmode} = \text{shutdown}) \land \\
(\text{newopmode} = \text{WasShutdown}(\text{opmode}, \text{selftestButton}, \text{waterlevel}, \\
\text{timeInMode}, \text{selftestButtonTime}) ) \\
\lor \\
(\text{opmode} = \text{standby}) \land \\
(\text{newopmode} = \text{WasInStandby}(\text{opmode}, \text{selftestButton}, \text{waterlevel}, \\
\text{resetButtonTime}, \text{selftestButtonTime}) ) \\
\lor \\
(\text{opmode} = \text{test}) \land \\
(\text{newopmode} = \text{WasInTest}(\text{opmode}, \text{timeInMode}) )
\end{align*}
\]

\[
\begin{align*}
((\text{failmode} = \text{allok}) \land \\
(\text{newfailmode} = \text{WasAlloc}(\text{failmode}, \text{controlUnit}, \text{levelDevice}, \\
\text{timeDevice})) \\
\lor \\
(\text{failmode} = \text{badlevdev}) \land \\
(\text{newfailmode} = \text{WasBadLevdev}(\text{failmode}, \text{controlUnit}, \text{levelDevice}, \\
\text{timeDevice})) \\
\lor \\
(\text{failmode} = \text{hardfail}) \land (\text{newfailmode} = \text{hardfail})
\end{align*}
\]
2.3.5.1 GetNextMode

This function determines the next operating mode and failure mode.
---

--- 2.3.5.2 Check Button Times

```plaintext
CheckButtonTimes(step : R, resetButton : buttonType,
                  selftestButton : buttonType,
                  oldtimes : buttonTimesType)
newtimes : buttonTimesType

pre true
post let mk-buttonTimesType(oldresetTime, oldselftestTime) =
    oldtimes in
let mk-buttonTimesType(newresetTime, newselftestTime) =
    newtimes in

((resetButton = pressed) => (newresetTime =
    oldresetTime + step)) ∧

((selftestButton = pressed) => (newselftestTime =
    oldselftestTime + step)) ∧

((resetButton = released) => (newresetTime = 0)) ∧

((selftestButton = released) => (newselftestTime = 0))
```

---

--- 2.3.5.3 GetInmodeTime :

--- This calculates the time spent in a given mode

```plaintext
GetInmodeTime(step : R, oldtime : R,
               oldmode : operatingModeType,
               newmode : operatingModeType) newtime : R

pre true
post (oldmode = newmode) ∧ (newtime = oldtime + step)
    ∨
    (oldmode ≠ newmode) ∧ (newtime = 0);
```
2.3.5.2 CheckButtonTimes

This function calculates the length of time the reset button and the selftest buttons have been pressed.

For each button, the current time step is added to the previous value, when the button is being pressed.

2.3.5.3 GetInmodeTime

This function calculates the time spent in the current mode.

If the mode has not changed, then the current time step is added to the previous time-in-mode value. If the mode has changed then the time-in-mode value is 0.
-- 2.3.5.4 Check Control Unit

CheckControlUnit(memory : deviceType, watchDog : watchDogType)
controlUnit : deviceType

pre true
post ((memory = ok) ∧ (watchDog = operate) ⇒ (controlUnit = ok)) ∧ ((memory = failed) ∨ (watchDog ∈ {uninit, shut}) ⇒ (controlUnit = failed));

-- 2.3.5.5 Check Level Device

CheckLevelDevice(dp : byte)
levelDevice : deviceType

pre true
post ((dp ∈ {0, 255}) ⇒ (levelDevice = failed)) ∧ (¬(dp ∈ {0, 255}) ⇒ (levelDevice = ok));
2.3.5.4 CheckControlUnit

This function checks the control unit.

If the memory has "failed" or the watchdog is either "shut" or "uninitialised" then the control unit is "failed", otherwise the control unit is "ok".

2.3.5.5 CheckLevelDevice

The level device is "failed" if the differential pressure is either 0 or 255, otherwise the level device is "ok".
2.3.5 Update Stored Data Function

UpdateStoredData(oldstore : storedDataType, timeNow : R,
waterlevel : R, resetButton : buttonType,
selftestButton : buttonType,
watchdog : watchDogType, memory : deviceType,
diffPress : byte, timeDevice : deviceType)
newstore : storedDataType

pre true
post let mk-storedDataType(time, timeInMode, watchdogTime,
modes, buttonTimes) = oldstore in
let mk-storedDataType(newtime, newtimeInMode, newwatchdogTime,
newmodes, newbuttonTimes) = newstore in
let step = timeNow - time in
let controlUnit = CheckControlUnit(memory, watchdog) in
let levelDevice = CheckLevelDevice(diffPress) in

(newtime = timeNow) \n
(newtimeInMode = GetInmodeTime(step, timeInMode,
modes.operatingMode, newmodes.operatingMode)) \n
(newwatchdogTime = step) \n
(newmodes = GetNextMode(waterlevel, resetButton, selftestButton,
oldstore, controlUnit, levelDevice, timeDevice)) \n
(newbuttonTimes = CheckButtonTimes(step, resetButton,
selftestButton, buttonTimes))\n;
2.3.5 UpdateStoredData

This function updates the values of the following stored data:

1. system time
2. time in mode
3. watchdog time
4. system modes
5. selftest button and reset button times
-- 2.3.6 Get Water Level

GetWaterLevel(dp : byte) waterlevel : R
pre true
post let level = dp / 255 x DSscale + DPOffset in
  (dp = 255) \land (waterlevel < levelLowerCal)
\lor
  (dp = 0) \land (waterlevel > levelUpperCal)
\lor
  (1 < dp) \land (dp < 254) \land
  (waterlevel = UTIL\textquotesingle Limit(levelLowerCal, level, levelUpperCal));

-- 2.3.7 Check Watchdog Timer

CheckTimer(time : R) watchdog : watchDogType
pre true
post (time < watchdogtimeout) \land (watchdog = operate)
\lor
  (time \geq watchdogtimeout) \land (watchdog = shut);
2.3.6 GetWaterLevel

This function calculates the water level from the differential pressure.

If the differential pressure is 0 or 255 then the water level is defined non deterministically.

2.3.7 CheckTimer

This function checks the watchdog timer.

If the watchdog time (i.e. the time between successive calls of the timer) exceeds the watchdog timeout then the watchdog is set to "shut". If the alternative holds, then the watchdog is set to "operate"
-- 2.3 Control State Variables Function

\[
\text{ControlStateVariables}(\text{monitored} : \text{monitorVarType}, \\
\quad \text{oldStore} : \text{storedDataType}, \\
\quad \text{oldVariables} : \text{storedVarType}) \\
\quad \text{controlled} : \text{controlVarType}
\]

\text{pre true}

\text{post let } \text{mk-storedDataType}(\text{time}, \text{timeInMode}, \text{watchdogTime}, \\
\quad \text{modes}, \text{buttonTimes}) = \text{oldStore} \text{ in}
\quad \text{let } \text{mk-storedVarType}(\text{oldHighWindow}, \text{oldLowWindow}, \text{oldAlarm}, \\
\quad \quad \text{oldSignal}) = \text{oldVariables} \text{ in}
\quad \text{let } \text{mk-controlVarType}(\text{crtData}, \text{controlSignals}, \text{storeNow}, \\
\quad \quad \text{variablesNow}) = \text{controlled} \text{ in}
\quad \text{let } \text{mk-crtDataType}(\text{highWindow}, \text{lowWindow}, \text{levelDisplay}) = \\
\quad \quad \text{crtData} \text{ in}
\quad \text{let } \text{mk-monitorVarType}(\text{diffPress}, \text{resetButton}, \text{selftestButton}, \\
\quad \quad \text{powerNow}, \text{memory}, \text{timeNow}, \text{timeDevice}) = \text{monitored} \text{ in}
\quad \text{let } \text{waterlevel} = \text{GetWaterLevel}(\text{diffPress}) \text{ in}
\quad \text{let } \text{mk-controlSignalsType}(\text{alarm}, \text{shutdownSignal}, \text{pumpSwitch}, \\
\quad \quad \text{watchdog}) = \text{controlSignals} \text{ in}

\begin{align*}
\text{watchdog} & = \text{CheckTimer}(\text{watchdogTime}) \wedge \\
\text{shutdownSignal} & = \text{PumpControl}(\text{oldSignal}, \text{resetButton}, \text{modes}) \wedge \\
\text{crtData} & = \text{GetDisplayData}(\text{oldHighWindow}, \text{oldLowWindow}, \text{waterlevel}, \\
\quad \text{time}, \text{timeInMode}, \text{modes}, \text{powerNow}) \wedge \\
\text{alarm} & = \text{AlarmControl}(\text{oldAlarm}, \text{waterlevel}, \text{timeInMode}, \text{modes}, \\
\quad \text{powerNow}) \wedge \\
\text{pumpSwitch} & = \text{PumpEnvironment}(\text{powerNow}, \text{shutdownSignal}, \text{watchdog}) \wedge \\
\text{storeNow} & = \text{UpdateStoredData}(\text{oldStore}, \text{timeNow}, \text{waterlevel}, \\
\quad \text{resetButton}, \text{selftestButton}, \text{watchdog}, \\
\quad \text{memory}, \text{diffPress}, \text{timeDevice}) \wedge \\
\text{variablesNow} & = \text{mk-storedVarType}(\text{highWindow}, \text{lowWindow}, \text{alarm}, \\
\quad \text{shutdownSignal});
\end{align*}
2.3 ControlStateVariables

This function calculates the system controlled state variables and stored variables.
-- 2.4 Send Control Signals Function

SendControlSignals(initControl : controlOutputType, controlSignals : controlSignalsType)
finalControl : controlOutputType

pre true
post let mk-controlSignalsType(alarm, shutdown, pump, watchdog) =
               controlSignals in
               let mk-controlOutputType(initShutsigs, initWatchdogsigs, initPumpsigs, initAlarmsigs) = initControl in
               let mk-controlOutputType(finalShutsigs, finalWatchdogsigs, finalPumpsigs, finalAlarmsigs) = finalControl in
               (finalShutsigs = initShutsigs ~ [shutdown]) \n               (finalPumpsigs = initPumpsigs ~ [pump]) \n               (finalAlarmsigs = initAlarmsigs ~ [alarm]) \n               (finalWatchdogsigs = initWatchdogsigs ~ [watchdog])

-- OPERATIONS

operations

-- 1.0 Initialise Operation

Initialise()

ext wr monitoredInputs : monitorInputType
               wr systemOutput : systemOutputType
               wr storedVariables : storedVarType
               wr storedData : storedDataType
2.4 SendControlSignals

This function appends the current shutdown signal, pump signal, alarm signal and watchdog signal to the corresponding signal sequence.

OPERATIONS

1. Initialise

This operation initialises the following:

- high window to "on"
- low window to "on"
- alarm to "silent"
- shutdown signal to "shutdown"
pre \((0 \leq initTime) \land (initTime \leq 5000)\)

post let \(mk\text{-}stored\text{Data}\text{Type}(\text{time}, \text{timeInMode}, \text{watch\text{Dog}}\text{Time},\) \\
\(\text{modes}, \text{button\text{Times}}) = \text{storedData} \) in \\
let \(mk\text{-}button\text{Times}\text{Type}(\text{reset\text{Time}}, \text{self\text{test\text{Time}}}) = \text{button\text{Times}} \) in \\
let \(mk\text{-}modes\text{Type}(\text{opmode}, \text{failmode}) = \text{modes} \) in \\
let \(mk\text{-}monitor\text{Input}\text{Type}(dp0, \text{rst}0, \text{slf}0, \text{powerseq}0, \text{mem}0, \text{timeseq}0,\) \\
\(\text{dev}0) = \text{monitored\text{Inputs}} \) in \\
let \(mk\text{-}monitor\text{Input}\text{Type}(dp, \text{rst}, \text{slf}, \text{powerseq}, \text{mem}, \text{timeseq}, \text{dev}) = \) \\
\(\text{monitored\text{Inputs}} \) in \\
let \(\text{controls} = mk\text{-}control\text{Output}\text{Type}([\text{shutdown}], [\text{uninit}],\) \\
\([\text{open}], [\text{silent}]]) \) in \\
let \(\text{screen} = \text{ScreenBackground} \uparrow \{ (18,10) \mapsto \text{fillchar},\) \\
\((18,30) \mapsto \text{fillchar} \} \) in \\

\((\text{stored\text{Variables}\.high\text{Window} = \text{on}}) \land\) \\
\((\text{stored\text{Variables}\.low\text{Window} = \text{on}}) \land\) \\
\((\text{stored\text{Variables}\.alarm = silent}) \land\) \\
\((\text{stored\text{Variables}\.shutdown\text{Signal} = \text{shutdown}}) \land\) \\
\((\text{system\text{Output}\.control\text{Outputs} = \text{controls}}) \land\) \\
\((\text{system\text{Output}\.display\text{Signals} = [\text{screen}]}) \land\) \\

\((\text{timeInMode} = 0) \land (\text{watch\text{Dog}}\text{Time} = 0) \land\) \\
\((\text{reset\text{Time} = 0}) \land (\text{self\text{test\text{Time}} = 0}) \land\) \\
\((\text{opmode} = \text{standby}) \land (\text{failmode} = \text{allok}) \land\) \\
\(\exists i:Z \cdot\) \\
\(\exists j:Z \cdot\) \\
\((\text{time} = \text{timeseq}0(j)) \land\) \\
\((0 \leq \text{time}) \land (\text{time} \leq \text{initTime}) \land\) \\
\(\forall k:Z \cdot\) \\
\((i \leq k) \land (k < j) \Rightarrow (\text{powerseq}0(k) = \text{on})\) \\
\) \land\) \\
\((k \geq j) \Rightarrow (\text{powerseq}(k-j) = \text{powerseq}0(k)) \land\) \\
\((\text{timeseq}(k-j) = \text{timeseq}0(k)))\)

;
The operation Initialise is complete once the power has been "on" for initTime.
2.0 Normal Operation

NormalOperation ()

ext wr monitoredInputs : monitorInputType
   wr storedData : storedDataType
   wr storedVariables : storedVarType
   wr systemOutput : systemOutputType

pre true
post let monitored = MonitorDevices(monitoredInputs) in
  let controlled = ControlStateVariables(monitored, storedData, storedVariables) in
  let mk-controlVarType(crtData, controlSignals, storeNow, variablesNow) = controlled in
  let mk-systemOutputType(initControl, initDisplay) = systemOutput in
  let mk-systemOutputType(finalControl, finalDisplay) = systemOutput in

  (finalControl = SendControlSignals(initControl, controlSignals)) \&
  (finalDisplay = UpdateCRT(initDisplay, crtData)) \&
  (storedData = storeNow) \&
  (storedVariables = variablesNow) \&
  (monitoredInputs = UpdateMonitoredInputs(monitoredInputs))

;

3.0 WLMS Monitor Operation

WLMS : () ⇝ ()

WLMS() △
  ( Initialise();
      while true do
         NormalOperation()
   )

end WLMSYSTEM
2. NormalOperation

This operation defines a single "read" of the monitored inputs and a single "write" of the controlled outputs. Stored data and stored variables are also calculated.

3. WLMS

This operation defines the entire monitoring system. The definition is given explicitly.
-- UTILITY FUNCTIONS MODULE
--

module UTIL

exports
  all

definitions

values
  'space' : Char = " ";
  'zero' : Char = "0"

types
  digitType = Z
    inv(x) △ x ∈ {0,...,9}

functions

--
--
-- U.1 MaxIntBelow :
-- This returns the Largest integer less than or equal to a given number
--

MaxIntBelow(x : R) n : Z
pre true
post (n ≤ x) ∧
  ∀i : Z -
  (i < x) ⇒ (i ≤ n);
Module Util

This module exports the following functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.1 MaxintBelow</td>
<td>B53</td>
</tr>
<tr>
<td>U.2 Limit</td>
<td>B55</td>
</tr>
<tr>
<td>U.3 round</td>
<td>B55</td>
</tr>
<tr>
<td>U.4 Inrange</td>
<td>B55</td>
</tr>
<tr>
<td>IntToChar</td>
<td>B55</td>
</tr>
<tr>
<td>U.5 GetDigitChar</td>
<td>B57</td>
</tr>
</tbody>
</table>

U.1 MaxIntBelow

This function calculates the largest integer less than or equal to a given real number.
Limit(min : R, x : R, max : R) limited : R
pre min ≤ max
post (x < min) \land (limited = min) 
   \lor 
   (x > max) \land (limited = max) 
   \lor 
   (min ≤ x) \land (x ≤ max) \land (limited = x);

round(z : R, p : Z) rounded : R
pre 0 ≤ p
post true;— i.e. is not yet defined;

InRange(min : R, x : R, max : R) ok : B
pre true
post ok = ((min < x) \land (x < max));

IntToChar(n : Z) c : Char
pre (0 ≤ n) \land (n ≤ 9)
post true;— i.e. is not yet defined
U.2 Limit
This function limits the imported value of x between min and max.

U.3 round
This function is undefined.

U.4 InRange
This function checks that a given real value is within a given range.

IntToChar
This function is not defined.
U.5 Get Digit Character Function

GetDigitChar(n : Z, level : R) c : Char
pre (1 ≤ n) ∧ (n ≤ 4) ∧
(0 ≤ level) ∧ (level ≤ 100.0) ∧
(round(level, 1) = level)

post ∃ d1 : digitType • ∃ d2 : digitType • ∃ d3 : digitType • ∃ d4 : digitType •
level = 100×d1 + 10×d2 + d3 + d4/10 ∧
{
(n = 1) ∧ (level ≥ 100.0) ∧ (c = IntToChar(d1))
∨
(n = 1) ∧ (level ≤ 99.9) ∧ (c = spacechar)
∨
(n = 2) ∧ (level ≥ 10.0) ∧ (c = IntToChar(d2))
∨
(n = 2) ∧ (level ≤ 9.9) ∧ (c = spacechar)
∨
(n = 3) ∧ (level ≥ 1.0) ∧ (c = IntToChar(d3))
∨
(n = 3) ∧ (level ≤ 0.9) ∧ (c = zerochar)
∨
(n = 4) ∧ (c = IntToChar(d4))
}
end UTIL
U.5 GetDigitChar

This function converts a given digit of a real number to a character. The real number must be in the range 0.0 to 100.0, and must not be given to more than one decimal place.

A leading digit equal to zero is converted to the space character.
PASCAL ANIMATION

This appendix contains a VAX-Pascal animation derived from the VDM specification. Unlike the VDM specification, the animation can be compiled and run under VAX/VMS 5.3. VAX-Pascal was chosen, because the features of VDM used could be represented easily. The animation provides an executable model of the the VDM. It does not have a graphical interface, but simply reads successive values of the monitored inputs from a file and outputs the corresponding values of the controlled variables.
{ 

WMMS ANIMATION
Version 1.0B  Feb 1991

}

program WMMSYSTEM(input, output, monitoredInputs, systemOutputs);
label
 999;

{

-------------------------------------------------------------

CONSTANTS

-------------------------------------------------------------

}

const
  spacechar = '"';
  levelLowerCal = 13.0;
  levelUpperCal = 27.0;
  shutdownLockTime = 200;
  watchdogtimeout = 500;
  waterTolerance = 0.5;
  hysteresis = 0.5;
  highWaterLimit = 26.0;
  lowWaterLimit = 14.0;
  LDscale = 11.1;
  maxAlarmTime = 4000;
  maxSelftestDelay = 500;
  maxResetDelay = 3000;
  minHWtestTime = 0;
  maxHWtestTime = 2000;
  minLWtestTime = 2000;

maxLWtestTime = 4000;
minLDTime = 4000;
maxLDTime = 13000;
maxTestDelay = 14000;
rowMin = 1;
rowMax = 25;
columnMin = 1;
columnMax = 40;

{

------------------------------------------------
TYPES
------------------------------------------------

}

type
    byte = 0..255;
    deviceType = (ok, failed);
    watchDogType = (uninit, operate, shut);
    onOffType = (on, off, undefined);
    buttonType = (pressed, released);
    operatingModeType = (undefOpMode, operating, shutdown, standby, test);
    failureModeType = (undefFailMode, allok, badlevdev, hardfail);
    rows = rowMin..rowMax;
    columns = columnMin..columnMax;
    gridpoint = record
        r : rows; c : columns;
    end;
    screenType = array[rows, columns] of Char;
    pumpSwitchType = (open, closed, undefpump);
    shutdownSignalType = (operate_, shutdown_, undefined_);
    alarmType = (silent, audible, undefined_alarm);
    buttonTimesType = record
        resetButtonTime : Real;
        selftestButtonTime : Real;
    end;
    modesType = record
        operatingMode : operatingModeType;
        failureMode : failureModeType;
    end;
storedVarType = record
  highWindow : onOffType;
  lowWindow : onOffType;
  alarm : alarmType;
  shutdownSignal : shutdownSignalType;
end;

storedDataType = record
  time : Real;
  timeInMode : Real;
  watchDogTime : Real;
  modes : modesType;
  buttonTimes : buttonTimesType;
end;

crtDataType = record
  highWindow : onOffType;
  lowWindow : onOffType;
  levelDisplay : Real;
end;

controlSignalsType = record
  alarm : alarmType;
  shutdownSignal : shutdownSignalType;
  pumpSwitch : pumpSwitchType;
  watchdog : watchDogType;
end;

monitorVarType = record
  diffPress : byte;
  resetButton : buttonType;
  selftestButton : buttonType;
  powerNow : onOffType;
  memory : deviceType;
  timeNow : Real;
  timeDevice : deviceType;
end;

controlVarType = record
  crtData : crtDataType;
  controlSignals : controlSignalsType;
  storeNow : storedDataType;
  variablesNow : storedVarType;
end;
Auxiliary Types:

\[
\text{animationType} = \text{(running, stepping)};
\]

\[
\begin{array}{ll}
\text{SYSTEM STATE} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{storedData} : \text{storedDataType}; \\
\text{storedVariables} : \text{storedVarType}; \\
\text{monitoredInputs} : \text{text}; \\
\text{systemOutputs} : \text{text}; \\
\text{DPscale, DOffset, initTime} : \text{Real}; \quad \{ \text{Global Constants} \} \\
\text{fillChar} : \text{Char}; \\
\text{animationMode} : \text{animationType}; \\
\text{undefinedLevel} : \text{Real} := 1.0E10;
\end{array}
\]

\[
\begin{array}{ll}
\text{Imported Functions from module UTIL} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{function MaxIntBelow(x : real) : integer;} \\
\text{extern;} \\
\end{array}
\]
function Limit(min: Real; x: Real; max: Real): Real;
extern;

t-function round(x: Real; p: integer): Real;
extern;

function InRange(min: Real; x: Real; max: Real): Boolean;
extern;

function GetDigitChar(n: Integer; level: Real): Char;
extern;

function ScreenBackground: screenType;
forward;

function GetWindowChar(window: onOffType): Char;
forward;

{

------------------------------------------------------------------------
UTILITY FUNCTIONS
------------------------------------------------------------------------

}
function mk_controlVarType(crtData : crtDataType;
            controlSignals : controlSignalsType;
            storeNow : storedDataType;
            variablesNow : storedVarType) :
            controlVarType;
var mk : controlVarType;
begi
mk.crtData := crtData;
mk.controlSignals := controlSignals;
mk.storeNow := storeNow;
mk.variablesNow := variablesNow;
mk_controlVarType := mk;
end;

function mk_storedVarType(highWindow, lowWindow : onOffType;
                           alarm : alarmType;
                           shutdownSignal : shutdownSignalType) :
                           storedVarType;
var mk : storedVarType;
begi
mk.highWindow := highWindow;
mk.lowWindow := lowWindow;
mk.alarm := alarm;
mk.shutdownSignal := shutdownSignal;
mk_storedVarType := mk;
end;
function mk_storedDataType(time, timeInMode, watchdogTime : Real;
  modes : modesType;
  buttonTimes : buttonTimesType) :
  storedDataType;
var mk : storedDataType;
begin
  mk.time := time;
  mk.timeInMode := timeInMode;
  mk.watchdogTime := watchdogTime;
  mk.modes := modes;
  mk.buttonTimes := buttonTimes;
  mk_storedDataType := mk;
end;

INPUT - OUTPUT

{ 2.0.1 }
procedure UpdateMonitoredInputs(var inputs : text);
begin
  readln(inputs);
  if eof(inputs) then goto 999;
end;

{ 2.1 }
function MonitorDevices(var inputs : text) : monitorVarType;
var monitored : monitorVarType;
begin
  with monitored do
  begin
    read(inputs, diffPress, resetButton, selftestButton, powerNow, memory, timeNow, timeDevice);
  end;

  writeln(chr(27), '[3;1H'); { reposition cursor for VT220 }
  writeln(' Time is ', timeNow:10:2);
writeIn('Differential Pressure = ', diffPress:0);
writeIn('reset button is ', resetButton);
writeIn('selftest button is ', selftestButton);
writeIn('power is ', powerNow);
writeIn('memory state is ', memory);
writeIn('time device state is ', timeDevice);
writeIn;
writeIn('System Outputs:-
);
writeIn;
writeIn(systemOutputs, ' Time is ', timeNow:10:2);
writeIn(systemOutputs, 'Differential Pressure = ', diffPress:0);
writeIn(systemOutputs, 'reset button is ', resetButton);
writeIn(systemOutputs, 'selftest button is ', selftestButton);
writeIn(systemOutputs, 'power is ', powerNow);
writeIn(systemOutputs, 'memory state is ', memory);
writeIn(systemOutputs, 'time device state is ', timeDevice);
writeIn(systemOutputs);
writeIn(systemOutputs, 'System Outputs:-
);
writeIn(systemOutputs);

end;
MonitorDevices := monitored;
end;

{ 2.2 }
procedure UpdateCRT(var screenOutput : text; crtData : crtDataType);
var hwchar, lwchar, digit1, digit2, digit3, digit4 : Char;
i: integer;
screen : screenType;
begin
with crtData do
begin
hwchar := GetWindowChar(highWindow);
lwchar := GetWindowChar(lowWindow);
digit1 := GetDigitChar(1, levelDisplay);
digit2 := GetDigitChar(2, levelDisplay);
digit3 := GetDigitChar(3, levelDisplay);
digit4 := GetDigitChar(4, levelDisplay);

{ The ScreenBackground function is a dummy function }

screen := ScreenBackground;
screen[18,10] := hwchar; screen[18,30] := lwchar;
screen[6,20] := digit1;
screen[6,21] := digit2;
screen[6,22] := digit3;
screen[6,24] := digit4;

{ Write out level display }
write('Level Display = ');
write(screenOutput, 'Level Display = ');
for i := 20 to 24 do
begin
  write(screen[6,i]);
  write(screenOutput, screen[6,i]);
end;
writeln; writeln(screenOutput);

{ Write out High and Low Window indicators }

write('HIGH = [', screen[18,10], ']' );
write(screenOutput, 'HIGH = [', screen[18,10], ']' );
write('LOW = [', screen[18,30], ']' );
write(screenOutput, 'LOW = [', screen[18,30], ']' );
writeln; writeln(screenOutput);
end;
end;

procedure SendControlSignals(var controlOutputs : text; controlSignals: controlSignalsType);
const width = 20;
begin
  with controlSignals do
  begin
    writeln(controlOutputs, 'SHUT DOWN SIGNAL = ', shutdownSignal : width);
    writeln(controlOutputs, 'WATCHDOG = ', watchdog : width);
    writeln(controlOutputs, 'PUMP SWITCH = ', pumpSwitch : width);
    writeln(controlOutputs, 'ALARM = ', alarm : width);
    writeln('SHUT DOWN SIGNAL = ', shutdownSignal : width);
    writeln('WATCHDOG = ', watchdog : width);
    writeln('PUMP SWITCH = ', pumpSwitch : width);
    writeln('ALARM = ', alarm : width);
  end;

  { Print out the next operating mode and failure mode }

with storedData do
begin
  with modes do
  begin
    writeln('Next Mode = (', operatingMode, ',', failuremode, ')');
    writeln;
    writeln(systemOutputs, 'Next Mode = (', operatingMode, ',', failuremode, ')');
    writeln(controlOutputs);
  end;
end;
end;

{ }

-----------------------------------------------
FUNCTIONS
-----------------------------------------------

{ 2.2.1 } function GetWindowChar; { (window : onOffType) : Char; }
var c : Char;
begin
  c := '?'; { i.e. initially undefined }
  if (window = off) then c := spacechar;
  if (window = on) then c := fillchar;

  GetWindowChar := c;
end;

{ 2.2.2 } function ScreenBackground; { : screenType; }
var screen : screenType;
begin
  screen[6,23] := 'a';
  screenBackground := screen;
end;
2.3.1 Pump Control Function

function PumpControl(oldSignal : shutdownSignalType;
  resetButton : buttonType; modes : modesType) :
  shutdownSignalType;

var shutdownSignal : shutdownSignalType;
  opmode : operatingModeType;
  failmode : failureModeType;
begin
  shutdownSignal := undefined_;  { initialise to the undefined state }
  opmode := modes.operatingMode;
  failmode := modes.failureMode;

  if (opmode = operating) and (failmode = allok) and
      not (resetButton = pressed) then shutdownSignal := operate_;

  if (opmode = operating) and (failmode = allok) and
      (resetButton = pressed) then shutdownSignal := oldSignal;

  if (opmode = shutdown) and (failmode = allok)
    then shutdownSignal := oldSignal;

  if (opmode in [standby, test]) and (failmode = allok)
    then shutdownSignal := shutdown_;

  if (failmode in [badlevdev, hardfail])
    then shutdownSignal := shutdown_;

  PumpControl := shutdownSignal;
end;
2.3.2.1 FlashOff Function

\[\text{function FlashOff(miliseconds : Real) : Boolean; }\]
\[\text{var residue : integer; }\]
\[\text{ flash : boolean; }\]
\[\text{begin }\]
\[\text{residue := MaxIntBelow(miliseconds/500) mod 2; }\]
\[\text{flash := (residue = 0); }\]
\[\text{FlashOff := flash; }\]
\[\text{end; }\]

2.3.2.2 Get High Window Function

\[\text{function GetHighWindow(oldHighWindow : onOffType; }\]
\[\text{ waterlevel, time, timelnMode : Real; }\]
\[\text{ modes : modesType; powerNow : onOffType) : }\]
\[\text{onOffType; }\]
\[\text{var highWindow : onOffType; }\]
\[\text{ opMode : operatingModeType; }\]
\[\text{ failMode : failureModeType; }\]
\[\text{ enter, ishigh, istesting : boolean; }\]
\[\text{begin }\]
\[\text{highWindow := undefined; \{ initialise to the undefined state }\}

\{
\text{ pre condition
\}
\text{ opMode := modes.operatingMode; }\]
\[\text{ failMode := modes.failureMode; }\]
\[\text{ if (powerNow = on) and not (failmode = hardfail) then }\]
\[\text{begin }\]
{ post condition }

ishigh := (waterlevel >= highWaterLimit);
enter := (timelnMode = 0);
istesting := (minHWtestTime <= timelnMode) and
        (timelnMode < maxHWtestTime);

if (opmode in [operating, shutdown]) and (failmode = allok) and
    ishigh
then highWindow := on;

if (opmode = operating) and (failmode = allok) and enter and
    not ishigh
then highWindow := off;

if (opmode = operating) and (failmode = allok) and not enter and
    not ishigh
then highWindow := oldHighWindow;

if (opmode = shutdown) and (failmode = allok) and not ishigh
then highWindow := oldHighWindow;

if (opmode = standby) and (failmode = allok)
then highWindow := oldHighWindow;

if (opmode = test) and (failmode = allok) and istesting
then highWindow := on;

if (opmode = test) and (failmode = allok) and not istesting
then highWindow := off;

if (failmode = badlevdev) and FlashOff(time) then highWindow := on;
if (failmode = badlevdev) and not FlashOff(time)
then highWindow := off;

end;
GetHighWindow := highWindow;
end;
2.3.2.3 Get Low Window Function

```plaintext
function GetLowWindow(oldLowWindow : onOffType;
    waterlevel, time, timeInMode : Real;
    modes : modesType; powerNow : onOffType) :
    onOffType;
var lowWindow : onOffType;
    opMode : operatingModeType;
    failMode : failureModeType;
    enter, islow, istesting : boolean;
begin
    lowWindow := undefined; { initialise to the undefined state }
{ pre condition }
    opMode := modes.operatingMode;
    failMode := modes.failureMode;
    if (powerNow = on) and not (failmode = hardfail) then begin
{ post condition }
        islow := (waterlevel <= lowWaterLimit);
        enter := (timelnMode = 0);
        istesting := (minLWtestTime <= timelnMode) and
            (timelnMode < maxLWtestTime);
        if (opmode in [operating, shutdown]) and (failmode = allok) and
            islow then lowWindow := on;
        if (opmode = operating) and (failmode = allok) and enter and
            not islow then lowWindow := off;
        if (opmode = operating) and (failmode = allok) and not enter and
            not islow then lowWindow := oldLowWindow;
    end;
end;
```

if (opmode = shutdown) and (failmode = allok) and not islow then lowWindow := oldLowWindow;

if (opmode = standby) and (failmode = allok) then lowWindow := oldLowWindow;

if (opmode = test) and (failmode = allok) and istesting then lowWindow := on;

if (opmode = test) and (failmode = allok) and not istesting then lowWindow := off;

if (failmode = badlevdev) and FlashOff(time) then lowWindow := off;

if (failmode = badlevdev) and not FlashOff(time) then lowWindow := on;

end;
GetLowWindow := lowWindow;
end;

{ 2.3.2.4 Get Level display Function
---------------------------
}
function GetLevelDisplay(waterlevel, time, timelnMode : Real;
modes : modesType) : Real;
var levelDisplay : Real;
opMode : operatingModeType;
failMode : failureModeType;
stage1, stage2, stage3 : boolean;
begin
levelDisplay := undefinedLevel; { initialise to an undefined state }

opMode := modes.operatingMode;
failMode := modes.failureMode;
stage1 := (0 <= timelnMode) and (timelnMode < minLDTime);
stage2 := (minLDTime <= timelnMode) and (timelnMode < maxLDTime);
stage3 := (timelnMode >= maxLDTime);
if failmode = allok then
  begin
    if (opmode in [operating, shutdown, standby])
      then levelDisplay := round(waterlevel,1);

    if (opmode = test) and stage1 then levelDisplay := 0;

    if (opmode = test) and stage2
      then levelDisplay := LDscale * MaxIntBelow((timeInMode-minLDTime)/1000);

    if (opmode = test) and stage3 then levelDisplay := 0;
  end;

GetLevelDisplay := levelDisplay;
end;

{
  2.3.2 Get Display Data Function
  -----------------------------
}

function GetDisplayData(oldHighWindow, oldLowWindow : onOffType;
  waterlevel, time, timeInMode : Real;
  modes : modesType; powerNow : onOffType) :
  crtDataType;
var crtData : crtDataType;
begin
  with crtData do
  begin
    highWindow := GetHighWindow(oldHighWindow, waterlevel, time,
      timeInMode, modes, powerNow);
    lowWindow := GetLowWindow(oldLowWindow, waterlevel, time,
      timeInMode, modes, powerNow);
    levelDisplay := GetLevelDisplay(waterlevel, time, timeInMode,
      modes);
    GetDisplayData := crtData;
  end;
end;
2.3.3 Alarm Control Function

```pascal
function AlarmControl(oldAlarm : alarmType;
  waterlevel, timelnMode : Real;
  modes : modesType; powerNow : onOffType) :
  alarmType;
var failMode : failureModeType;
  opMode : operatingModeType;
  alarm : alarmType;
  isinlimits, isalarm, enter : boolean;
begins
  alarm := undefined_alarm; { initialise to the undefined state }
  { pre condition }
  opMode := modes.operatingMode;
  failMode := modes.failureMode;
  if (powerNow = on) and not (failmode = hardfail) then begin
    { post condition }
    isinlimits := InRange(lowWaterLimit, waterlevel, highWaterLimit);
    isalarm := (0 <= timelnMode) and (timelnMode < maxAlarmTime);
    enter := (0 = timelnMode);
    if (opmode in [shutdown, operating]) and (failmode = allok) and
      not isinlimits then alarm := audible;
    if (opmode = operating) and (failmode = allok) and enter and
      isinlimits then alarm := silent;
    if (opmode = operating) and (failmode = allok) and not enter and
      isinlimits then alarm := oldAlarm;
    if (opmode = shutdown) and (failmode = allok) and
      isinlimits then alarm := oldAlarm;
    if (opmode = standby) and (failmode = allok) then alarm := oldAlarm;
```

if (opmode = test) and (failmode = allok) and
  isalarm then alarm := audible;

if (opmode = test) and (failmode = allok) and
  not isalarm then alarm := silent;

if (failmode = badlevdev) and enter then alarm := audible;

if (failmode = badlevdev) and not enter then alarm := oldAlarm;
end;
AlarmControl := alarm;
end;

{ 2.3.4 Pump Environment Function
  ------------------------
}
function PumpEnvironment(powerNow: onOffType;
  shutdownSignal: shutdownSignalType;
  watchDog: watchDogType):
pumpSwitchType;
var pumpSwitch: pumpSwitchType;
begin
  pumpSwitch := undefpump;  { initially undefined }

  if (powerNow = on) and (shutdownSignal = operate) and
      (watchDog = operate) then pumpSwitch := closed;

  if (powerNow = off) or (shutdownSignal = shutdown) or
     (watchDog = shut) then pumpSwitch := open;

  PumpEnvironment := pumpSwitch;
end;
2.3.5.1.1 Was in Operating Mode

function WasOperating(initmode: operatingModeType;
selftestButton: buttonType;
waterlevel, selftestButtonTime: Real): operatingModeType;

var mode: operatingModeType;
    inshut, intest: boolean;
begin
    mode := undefOpMode;
    if initmode = operating then
        begin
            inshut := (selftestButton <> pressed) and
                not InRange(lowWaterLimit, waterlevel, highWaterLimit);
            intest := (selftestButton = pressed) and
                (selftestButtonTime >= maxSelftestDelay);
            if inshut then mode := shutdown;
            if intest then mode := test;
            if (not intest and not inshut) then mode := operating;
        end;
    WasOperating := mode;
end;

2.3.5.1.2 Was in Shutdown Mode

function WasShutdown(initmode: operatingModeType;
selftestButton: buttonType;
waterlevel, timelnMode,
selftestButtonTime: Real): operatingModeType;

var mode: operatingModeType;
    isinhyst, inop1, instnd1, intest: boolean;
begin
    mode := undefOpMode;
    if initmode = shutdown then
begin
isinyst := InRange(lowWaterLimit + hysteresis, waterlevel,
                    highWaterLimit-hysteresis);
inop1 := (selftestButton <> pressed) and
       (timelnMode < shutdownLockTime) and isinyst;
instd1 := (selftestButton <> pressed) and
       (timelnMode >= shutdownLockTime);
 intest := (selftestButton = pressed) and
       (selftestButtonTime >= maxSelftestDelay);

if inop1 then mode := operating;
if instnd1 then mode := standby;
if intest then mode := test;
if (not inop1 and not instnd1 and not intest)
then mode := shutdown;
end;
WasShutdown := mode;
end;

{  
  2.3.5.1.3 Was in Standby Mode
  -------------------
}
function WasinStandby(initmode : operatingModeType;
                        selftestButton : buttonType; waterlevel,
                        resetButtonTime, selftestButtonTime : Real) :
                        operatingModeType;
var mode : operatingModeType;
isinhyst, inop2, intest : boolean;
begin
mode := undefOpMode;
if initmode = standby then
begin
isinyst := InRange(lowWaterLimit + hysteresis, waterlevel,
                    highWaterLimit-hysteresis);
inop2 := (resetButtonTime >= maxResetDelay) and isinyst;
intest := (selftestButton = pressed) and
       (selftestButtonTime >= maxSelftestDelay);
if inop2 then mode := operating;
if intest then mode := test;
if (not inop2 and not intest) then mode := standby;
end;

WasinStandby := mode;
end;

2.3.5.1.4 Was in Test Mode
----------

function WasinTest(initmode : operatingModeType;
                    timelnMode : Real) : operatingModeType;
var mode : operatingModeType;
    instnd2 : boolean;
begin
    mode := undefOpMode;
    if initmode = test then
        begin
            instnd2 := (timelnMode >= maxTestDelay);
            if instnd2 then mode := standby;
            if not instnd2 then mode := test;
        end;
    WasinTest := mode;
end;
function WasAllok(initmode : failureModeType;
    controlUnit, levelDevice, timeDevice : deviceType) :
    failureModeType;
var mode : failureModeType;
    lev, hrd : boolean;
begin
    mode := undefFailMode;
    if initmode = allok then
begin
        hrd := (controlUnit = failed) or (timeDevice = failed);
        lev := (levelDevice = failed) and (controlUnit = ok) and
            (timeDevice = ok);
        if lev then mode := badlevdev;
        if hrd then mode := hardfail;
        if (not lev and not hrd) then mode := allok;
    end;
    WasAllok := mode;
end;
function WasBadLevdev(initmode : failureModeType;
    controlUnit, levelDevice,
    timeDevice : deviceType) : failureModeType;

var mode : failureModeType;
    hrd : boolean;
begin
    mode := undefFailMode;
    if initmode = badlevdev then
    begin
        hrd := (controlUnit = failed) or (timeDevice = failed);

        if hrd then mode := hardfail;

        if not hrd then mode := badlevdev;
    end;

    WasBadLevdev := mode;
end;
2.3.5.1 Get Next Mode

function GetNextMode(waterlevel : Real; resetButton, selftestButton : buttonType; store : storedDataType; controlUnit, levelDevice, timeDevice : deviceType) : modesType;

var newmodes : modesType;
failMode : failureModeType;
opMode : operatingModeType;
newfailMode : failureModeType;
newopMode : operatingModeType;

begin
  with store do
    begin
      with modes, buttonTimes do
        begin
          opMode := operatingMode;
          failMode := failureMode;

          if (opmode = operating) then
            newopmode := WasOperating(opmode, selftestButton, waterlevel, resetButtonTime)
          else if (opmode = shutdown) then
            newopmode := WasShutdown(opmode, selftestButton, waterlevel, timelnMode, selftestButtonTime)
          else if (opmode = standby) then
            newopmode := WasinStandby(opmode, selftestButton, waterlevel, resetButtonTime, selftestButtonTime)
          else if (opmode = test) then
            newopmode := WasinTest(opmode, timelnMode);

          if (failmode = alllok) then
            newfailmode := WasAlllok(failmode, controlUnit, levelDevice, timeDevice)
          else if (failmode = badlevdev) then
            newfailmode := WasBadLevdev(failmode, controlUnit, levelDevice, timeDevice)
        end
    end
end
else if (failmode = hardfail) then
    newfailmode := hardfail;
end;
end;

newmodes.operatingMode := newopmode;
newmodes.failureMode := newfailmode;
GetNextMode := newmodes;
end;

2.3.5.2 Check Button Times
--------------------------

}  

function CheckButtonTimes(step : Real; resetButton, selftestButton : buttonType; 
oldtimes : buttonTimesType) : buttonTimesType;
var newtimes : buttonTimesType;
oldresetTime, oldselftestTime,
newresetTime, newselftestTime : Real;
begin
with oldtimes do 
begin
oldresetTime := resetButtonTime;
oldselftestTime := selftestButtonTime;

if (resetButton = pressed) then 
    newresetTime := oldresetTime + step;

if (selftestButton = pressed) then 
    newselftestTime := oldselftestTime + step;

if (resetButton = released) then newresetTime := 0;

if (selftestButton = released) then newselftestTime := 0;
end;

newtimes.resetButtonTime := newresetTime;
newtimes.selftestButtonTime := newselftestTime;
CheckButtonTimes := newtimes;
end;
2.3.5.3 GetlnmodeTime :
---------------------

function GetlnmodeTime(step, oldtime : Real;
    oldmode, newmode : operatingModeType) : Real;
var newtime : Real;
begin
    if (oldmode = newmode) then newtime := oldtime + step
    else if (oldmode <> newmode) then newtime := 0;
    GetlnmodeTime := newtime;
end;

2.3.5.4 Check Control Unit
-------------------------

function CheckControlUnit(memory : deviceType;
    watchDog : watchDogType) : deviceType;

var controlUnit : deviceType;
begin
    if (memory = ok) and (watchDog = operate) then controlUnit := ok;

    if ((memory = failed) or (watchDog in [uninit, shut])) then
        controlUnit := failed;

    CheckControlUnit := controlUnit;
end;
2.3.5.5 Check Level Device
---------------------

function CheckLevelDevice(dp : byte) : deviceType;
var levelDevice : deviceType;
begin
  if (dp in [0, 255]) then levelDevice := failed;
  if not (dp in [0, 255]) then levelDevice := ok;
  CheckLevelDevice := levelDevice;
end;

2.3.5 Update Stored Data Function
---------------------------------

function UpdateStoredData(oldstore : storedDataType; timeNow, waterlevel : Real; resetButton, selftestButton : buttonType; watchdog : watchdogType; diffPress : byte; memory, timeDevice : deviceType) : storedDataType;
var newstore : storedDataType;
  step, newtime, newtimeInMode, newwatchdogTime : Real;
  newbuttonTimes : buttonTimesType;
  newmodes : modesType;
  controlUnit, levelDevice : deviceType;
begin
  controlUnit := CheckControlUnit(memory, watchdog);
  levelDevice := CheckLevelDevice(diffPress);
  with oldstore do
    begin
      step := timeNow - time;
newtime := timeNow;

newmodes := GetNextMode(waterlevel, resetButton, selftestButton, oldstore, controlUnit, levelDevice, timeDevice);

newtimeInMode := GetInmodeTime(step, timelnMode, modes.operatingMode, newmodes.operatingMode);

newwatchdogTime := step;

newbuttonTimes := CheckButtonTimes(step, resetButton, selftestButton, buttonTimes);

end;

newstore := mk_storedDataType(newtime, newtimeInMode, newwatchdogTime, newmodes, newbuttonTimes);

UpdateStoredData := newstore;
end;

2.3.6 Get Water Level
--------------------

function GetWaterLevel(dp : byte) : Real;
var waterlevel, level : Real;
begin
  level := dp; { i.e. convert from byte to Real }
  level := level / 255 * DPscale + DPOffset;
  if (dp = 255) then
    waterlevel := levelLowerCal - 0.5 { i.e. < levelLowerCal }
  else if (dp = 0) then
    waterlevel := levelUpperCal + 0.5 { i.e. > levelUpperCal }
  else if (1 <= dp) and (dp <= 254) then
    waterlevel := Limit(levelLowerCal, level, levelUpperCal);
  GetWaterLevel := waterlevel;
end;
2.3.7 Check Watchdog Timer

```pascal
function CheckTimer(time : Real) : watchdogType;
var watchdog : watchdogType;
begin
  if (time < watchdogtimeout) then watchdog := operate
  else if (time >= watchdogtimeout) then watchdog := shut;
  CheckTimer := watchdog;
end;
```

2.3 Control State Variables Function

```pascal
function ControlStateVariables(monitored : monitorVarType;
oldStore : storedDataType;
oldVariables : storedVarType) :
controlVarType;
var controlled : controlVarType;
  waterlevel : real;
  crtData : crtDataType;
  controlSignals : controlSignalsType;
  storeNow : storedDataType;
  variablesNow : storedVarType;
  oldHighWindow, oldLowWindow : onOffType;
  oldAlarm : alarmType;
  oldSignal : shutdownSignalType;
begin
  with oldVariables do
  begin
    oldHighWindow := highWindow;
    oldLowWindow := lowWindow;
```
oldAlarm := alarm;
oldSignal := shutdownSignal;
end;

with monitored, storedData, controlSignals do
begin
  waterlevel := GetWaterLevel(diffPress);

  alarm := AlarmControl(oldAlarm, waterlevel, timelnMode, modes, powerNow);

  shutdownSignal := PumpControl(oldSignal, resetButton, modes);

  controlSignals.watchdog := CheckTimer(watchdogTime);

  pumpSwitch := PumpEnvironment(powerNow, shutDownSignal, watchdog);

  crtData := GetDisplayData(oldHighWindow, oldLowWindow, waterlevel, time, timelnMode, modes, powerNow);

  storeNow := UpdateStoredData(oldStore, timeNow, waterlevel, resetButton, selftestButton, watchdog, diffPress, memory, timeDevice);

  variablesNow := mk_storedVarType(crtData.highWindow, crtData.lowWindow, alarm, shutdownSignal);

end;

controlled := mk_controlVarType(crtData, controlSignals, storeNow, variablesNow);

ControlStateVariables := controlled;
end;
{ ANIMATION SETUP }

procedure Animation_setup;
begin
  fillchar := 'X'; \{ i.e. is not yet defined \}
  DOpool := 1.03803 * (levelUpperCal - levelLowerCal);
  DPoffset := -0.01902 * (levelUpperCal - levelLowerCal) +
               levelLowerCal;
  reset(monitoredInputs);
  rewrite(systemOutputs);
  writeln(chr(12), chr(12));
  writeln(chr(27), '[1;1H');
  writeln(chr(27), '#6 ANIMATION');
  writeln(chr(27), '[3;1H');

  writeln('Type RUNNING or STEPPING');

  animationMode := readTerminal;

end;

procedure nextAnimationStep;
begin
  if animationMode = stepping then
  begin
    writeln('Type RETURN to continue >>');
    readln;
  end
end;
procedure Initialise;
var ready : boolean;
    monitored : monitorVarType;
begin
    writeln('START OF INITIALISATION');
    writeln(systemOutputs,'START OF INITIALISATION');

    if (0 <= initTime) and (initTime <= 5000) then begin
        storedVariables.highWindow := on;
        storedVariables.lowWindow := on;
        storedVariables.alarm := silent;
        storedVariables.shutdownSignal := shutdown;

        storedData.timelnMode := 0;
        storedData.watchdogTime := 0;
        storedData.modes.operatingMode := standby;
        storedData.modes.failureMode := alloc;
        with storedData.buttonTimes do begin
            resetButtonTime := 0;
            selftestButtonTime := 0;
        end;

        repeat
            monitored := MonitorDevices(monitoredInputs);
            UpdateMonitoredInputs(monitoredInputs);

            with monitored do begin
                ready := (0 <= timeNow) and (timeNow <= initTime) and
                        (powerNow = on);
                storedData.time := timeNow;
            end;
        until ready;
    end;
write('HIGH = [', fillchar, ']');
write(systemOutputs, 'HIGH = [', fillchar, ']');
write('LOW = [', fillchar, ']');
write(systemOutputs, 'LOW = [', fillchar, ']');
writeln;
writeln(systemOutputs);
writeln('SHUT DOWN SIGNAL = ', shutdown_);
writeln(systemOutputs, 'SHUT DOWN SIGNAL = ', shutdown_);
writeln('WATCHDOG = ', uninit);
writeln(systemOutputs, 'WATCHDOG = ', uninit);
writeln('PUMP SWITCH = ', open);
writeln(systemOutputs, 'PUMP SWITCH = ', open);
writeln('ALARM = ', silent);
writeln(systemOutputs, 'ALARM = ', silent);
writeln;
writeln(systemOutputs);
writeln('------ END OF INITIALISATION ------');
writeln(systemOutputs, '------ END OF INITIALISATION ------');

nextAnimationStep;

writeln(chr(12), chr(12));
writeln(systemOutputs);
writeln(systemOutputs);

end
else begin
writeln(systemOutputs, 'INITIALISE PRECONDITION = FALSE');
goto 999;
end;
end;

procedure NormalOperation;
var controlled : controlVarType;
  monitored : monitorVarType;
begin

  monitored := MonitorDevices(monitoredInputs);

  controlled := ControlStateVariables(monitored, storedData, storedVariables);
with controlled do
begin
  storedData := storeNow;
  storedVariables := variablesNow;
  UpdateCRT(systemOutputs, crtData);
  SendControlSignals(systemOutputs, controlSignals);
end;

UpdateMonitoredInputs(monitoredInputs);
end;

begin
  Animation_setup;

{  
  3.0 WLMS Monitor Operation 

}  
Initialise;

while true do
begin
begin
  NormalOperation;

  { Control the Animation }
  nextAnimationStep;
end;

{  
  QUIT ANIMATION 

}  
999:
end.
MODULE UTIL
Version 1.0A  Feb 1991

module UTIL(output);

const
  spacechar = ' ';  
  zerochar = '0';

type
digitType = 0..9;

{ }
[global] function round(x : Real; p : integer) : Real;
  var scale : Real;
  begin
    scale := 10**p;
    round := trunc(x * scale + 0.5) / scale;
  end;

[global] functionInRange(min : Real; x : Real; max : Real) : Boolean;
  begin
    InRange := (min < x) and (x < max);
  end;

function IntToChar(n : Integer) : Char;
  begin
    if (0 <= n) and (n <= 9) then
      begin
        IntToChar := chr(n + ord('0'));
      end
    else IntToChar := '?';
  end;
[global] function GetDigitChar(n: Integer; level: Real): Char;
var d1, d2, d3, d4: integer;
    c: Char;
begin
    d1 := trunc(level / 100);
    d2 := trunc((level / 10) - 10 * d1);
    d3 := trunc(level) - 100 * d1 - 10 * d2;
    d4 := trunc((10 * level) - 1000 * d1 - 100 * d2 - 10 * d3);
    c := '?';
    if (level <= 999.9) then
        begin
            if (n = 1) and (level >= 100.0) then c := IntToChar(d1)
            else if (n = 1) and (level < = 99.9) then c := spacechar
            else if (n = 2) and (level >= 10.0) then c := IntToChar(d2)
            else if (n = 2) and (level < = 9.9) then c := spacechar
            else if (n = 3) and (level >= 1.0) then c := IntToChar(d3)
            else if (n = 3) and (level < = 0.9) then c := zerochar
            else if (n = 4) then c := IntToChar(d4);
        end;
    GetDigitChar := c;
end;
end. { of module UTIL }
EXAMPLE ANIMATION RUN

1. **Animation Input Format**

The animation reads data from a VAX file named "monitoredinputs.dat". This contains the data for each main program cycle in the following format:

\[
\text{diffPress, reset, selftest, power, memory, time, timeDevice}
\]

where

- **diffPress** - Differential pressure input i.e. an integer in the range 0 to 255
- **reset** - The state of the reset button i.e. pressed or released
- **selftest** - The state of the selftest button i.e. pressed or released
- **power** - System power i.e. on or off
- **memory** - The state of the computer memory i.e. ok or failed
- **time** - Current time i.e. a real number
- **timeDevice** - State of the time device i.e. ok or failed

The following is a typical example of an input data file for the animation:

\[
155 \text{ released released on ok } 0 \text{ ok} \\
156 \text{ released released on ok } 250 \text{ ok} \\
157 \text{ released released on ok } 500 \text{ ok} \\
158 \text{ released released on ok } 750 \text{ ok} \\
159 \text{ released released on ok } 1000 \text{ ok}
\]
2. **Animation Output Format**

Each program cycle, the animation displays the inputs and outputs to both the screen and to the VAX file "systemoutputs.dat". The following is typical of the output produced by the animation:

```
Time is 500.00
Differential Pressure = 155
reset button is RELEASED
selftest button is RELEASED
power is ON
memory state is OK
time device state is OK

System Outputs:-

Level Display = 21.6
HIGH = [X] LOW = [X]
SHUT DOWN SIGNAL = SHUTDOWN
WATCHDOG = OPERATE
PUMP SWITCH = OPEN
ALARM = SILENT
Next Mode = ( STANDBY, ALLOK)
```
GLOSSARY

\{t_1,t_2,...,t_n\} \hspace{1cm} \text{set enumeration}
\{\} \hspace{1cm} \text{empty set}
\{i,...,j\} \hspace{1cm} \text{subset of integers}
t \in S \hspace{1cm} \text{set membership}
S \subseteq T \hspace{1cm} \text{subset of}
S \subset T \hspace{1cm} \text{strict subset of}
S \cap T \hspace{1cm} \text{set intersection}
S \cup T \hspace{1cm} \text{set union}
\o \hspace{1cm} \text{set generator}

Maps
M is a map
\text{domM} \hspace{1cm} \text{domain}
\text{rngM} \hspace{1cm} \text{range}
\{d_1 \mapsto r_1,...,d_n \mapsto r_n\} \hspace{1cm} \text{map enumeration}
\{\} \hspace{1cm} \text{empty map}
M(d) \hspace{1cm} \text{application}
M1 \upharpoonright M2 \hspace{1cm} \text{overwriting}

Sequences
s, t are sequences
\text{len s} \hspace{1cm} \text{length}
\text{hd s} \hspace{1cm} \text{head}
\text{tl s} \hspace{1cm} \text{tail}
[t_1, t_2,...,t_n] \hspace{1cm} \text{sequence enumerator}
\o \hspace{1cm} \text{empty sequence}
s \upharpoonright t \hspace{1cm} \text{concatenation}
s(i) \hspace{1cm} \text{sequence element}

Composite Objects
o is a composite object
:: \hspace{1cm} \text{compose}
\text{mk} - N() \hspace{1cm} \text{generator}
s(o) \hspace{1cm} \text{selector}

Logic
E_i \text{ are logical expressions}
\neg E \hspace{1cm} \text{negation}
E_1 \land E_2 \hspace{1cm} \text{conjunction}
E_1 \lor E_2 \hspace{1cm} \text{disjunction}
E_1 \Rightarrow E_2 \hspace{1cm} \text{implication}
E_1 \Leftrightarrow E_2 \hspace{1cm} \text{equivalence}
\forall x \in T . E \hspace{1cm} \text{universal quantifier}
\exists x \in T . E \hspace{1cm} \text{existential quantifier}
\Gamma \vdash E \hspace{1cm} \text{sequent E can be proved from } \Gamma
\Gamma \models E \hspace{1cm} \text{sequent E is true in all}
\hspace{1cm} \text{worlds in which } \Gamma \text{ holds}
Basic Sets

\[ \begin{align*}
N &\quad \{1,2,\ldots\} \\
B &\quad \{\text{true, false}\} \\
Z &\quad \{\ldots, -1, 0, 1, \ldots\} \\
R &\quad \text{real numbers} \\
\text{Char} &\quad \text{characters} \\
T_1 \times T_2 &\quad \text{product type} \\
T_1 \rightarrow T_2 &\quad \text{map type} \\
\text{inv}(T) \triangleq &\quad \text{type invariant definition}
\end{align*} \]

Functions

\[ \begin{align*}
f : D_1 \times D_2 &\rightarrow R &\text{signature} \\
f(d) &\text{application} \\
\text{let } x = \ldots \text{ in } \ldots &\text{local definition}
\end{align*} \]

Implicit Function Specification

\[ \begin{align*}
f(d : D) &\quad r : R &\text{signature} \\
\text{pre } \ldots d \ldots &\text{precondition} \\
\text{post } \ldots d \ldots r \ldots &\text{postcondition}
\end{align*} \]

Implicit Operation Specification

\[ \begin{align*}
\text{OP}(p : T_p) &\quad r : T_r &\text{signature} \\
\text{ext rd e_1 : } T_1, \ wr e_2 : T_2 &\text{external statement} \\
\text{pre } \ldots p \ldots e_1 \ldots e_2 \ldots &\text{precondition} \\
\text{post } \ldots p \ldots e_1 \ldots e_2 \ldots r \ldots e_2 \ldots &\text{postcondition}
\end{align*} \]