Master thesis report:

SYSTEM ANALYSIS APPLIED TO INTERFACE SYSTEMS

By: J.B. Slob
Supervisor: Prof. Ir. M.P.J. Stevens
Coach: Ir. F.P.M. Budzelaar

Eindhoven University of Technology,
Department of Electrical Engineering,
Digital Systems Group

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of graduation reports.
ABSTRACT

This master thesis is the result of my graduation project at the Eindhoven University of Technology. System analysis is mainly thought of as being used in the field of software engineering. My graduation project consisted of evaluating the use of system analysis on digital systems. Interface systems such as a queue and a Universal Asynchronous Receiver/Transmitter (UART) are modeled with the system analysis method.

For reasons of clarity a new type of data flow, the protocol data flow, is introduced and used in modelling the systems.

System analysis produces a comprehensive model of the interface system it is applied to.

ACKNOWLEDGEMENTS

My heartfelt thanks to Prof. Stevens. He clearly demonstrated that a professor not only is a "technology manager" but also a "human resources manager".

My thanks to my coach, Ir. F. Budzelaar for the advice and the fruitful discussions.

And regards to Ing. P. van der Putten for making timely observations.
FIGURE LIST

fig. 2.1: Transformations 1
fig. 2.2: Flows 2
fig. 2.3: Stores 3
fig. 2.4: Context Schema 4
fig. 2.5: Control Transformation 5
fig. 2.6: State Transition Diagram 6

fig. 3.1: Transformations with Store 8
fig. 3.2: Two better store notations 9
fig. 3.3: New notation for a store 9
fig. 3.4: Protocol Data Flows 10
fig. 3.5: Protocol Data Flow Decomposition 11

fig. 4.1: Queue memory with pointers 13
fig. 4.2: Queue memory full or empty 13
fig. 4.3: Queue Context Schema 15
fig. 4.4: Transformation Schema 17
fig. 4.5: State Transition Diagram 18
fig. 4.6: Queue Context Schema Level 0 21
fig. 4.7: Transformation Schema Level 0 21
fig. 4.8: Context Schema ???? 22
fig. 4.9: Context Schema Polled Mode 22
fig. 4.10: Transformation Schema of Queue in Polled Mode 23
fig. 4.11: State Diagram Polled Mode 24
fig. 4.12: Context Schema Event Mode 27
fig. 4.13: Transformation Schema of Queue in Event Mode 28

fig. 5.1: Context Schema with Protocol Flows 31
fig. 5.2: Connection between the MFIoC and the UART 32
fig. 5.3: MFIoC fully interlocked input mode 32
fig. 5.4: UART Transmitter 33
fig. 5.5: Connection MFIoC to UART with store 33
fig. 5.6: Connection MFIoC to UART with transformation 34
fig. 5.7: State Transition Diagram 34
fig. 5.8: Total Interconnections 36
fig. 6.1: UART used in interrupt mode 37
fig. 6.2: UART used in polled mode 37
fig. 6.3: Total System Interconnections 38
fig. 6.4: UART Context Schema 38
fig. 6.5: Context Schema with decomposed Protocol Flows 39
fig. 6.6: Level 0 Transformation Schema 40
fig. 6.7: Receiver Transformation Schema 41
fig. 6.8: Transmitter Transformation Schema 41
fig. 6.9: Receiver and Transmitter State Transition Diagrams 42
CONTENTS

1 INTRODUCTION 1

2 DESCRIPTION OF THE WARD/MELLOR REAL-TIME METHOD 2
  2.1 Basic Notations 2
     2.1.1 Transformations 2
     2.1.2 Flows 2
     2.1.3 Stores 3
  2.2 Elements of the Ward/Mellor Method 4
     2.2.1 Context Schema 4
     2.2.2 Event List 4
     2.2.3 Transformation Schema 5
     2.2.4 State Transition Diagram for the Control Transformations 5
     2.2.5 Description of the Data Transformations 6
     2.2.6 Data Dictionary 7

3 PROTOCOL DATA FLOWS 8
  3.1 Stores and their Dataflow 8
  3.2 Decomposition of Protocol Data Flows 10

4 QUEUE 12
  4.1 General Queue Description 12
  4.2 Queue with Fully Interlocked Handshake Protocol 14
     4.2.1 Context Schema 15
     4.2.2 Event List 15
     4.2.3 Transformation Schema 16
     4.2.4 State Transition Diagram for the Control Transformations 16
     4.2.5 Description of the Data Transformations 19
     4.2.6 Data Dictionary 20
  4.3 Queue Models with Protocol Data Flows 21
     4.3.1 Level 0 Model 21
     4.3.2 Queue in Polled Mode 22
     4.3.3 Queue in Event Mode 27
  4.4 Evaluation of the Queue Models 30

5 TOTAL INTERFACE SYSTEM DEVICE INTERCONNECTION 31
1 INTRODUCTION

As systems become more and more complex methods are needed to be able to manage and represent this greater complexity. Software engineering addresses the problems due to this complexity and tries to offer methods to solve the problems. System analysis (part of the software engineering field) is the examination, identification, and evaluation of the components and interrelationships involved in systems.

A system analysis method is described in Structured Development of Real Time Systems [1]. In it the authors describe a set of tools to model a real time system. This method (from now on referred to as the Ward/Mellor real-time method) is an extension to the data flow diagram. It adds timing and control information, making it possible to represent the interaction between the timing and control aspects of a system and its data transformation behaviour. It provides a notation and formulation rules for building a comprehensive system model. The Ward/Mellor real-time method uses a graphic modelling language, with a textual subset.

The modeled system, using the Ward/Mellor real time method, will consist of the following elements:
- context schema
- event list
- transformation schema
- state transition diagram for the control transformation
- description of the data transformations
- data dictionary

The object of this report is to see whether the Ward/Mellor real-time method is a usable method for modelling and the specification of digital systems. This was tested by modelling devices with the method. The interface devices that are modeled are a queue and a UART. The reports by Richter [2] and Stevens/Budzelaar [5] were partly used to make the models.
2 THE WARD/MELLOR REAL-TIME METHOD

In this chapter a brief description of the Ward/Mellor real-time method will be given. For a thorough description of the method the reader should consult the references. [1],[4]

2.1 Basic Notations

The transformation schema and part of the context schema make use of the following notations.

2.1.1 Transformations

Transformations are represented by circles. A solid circle represents a data transformation and a dotted circle a control transformation. (see fig. 2.1) The name that describes the function of the transformation is written inside the circle.

![Figure 2.1:Transformations](image)

A data transformation is an abstraction of data manipulation or stored data access.

A control transformation is an abstraction on some portion of the system's control logic. It controls the system by activating the data transformations. The control itself is described by the control transformation's state transition diagram.

2.1.2 Flows

Flows are represented by lines with arrow heads. For all types of flows, the way the arrow points defines the receiver. The flows are labelled by writing their names along them.
Discrete data flows are associated with a variable or set of variable values that are defined at discrete points in time. It is an abstraction on data sent or received as a unit by a system.

Continuous data flows are associated with a value or set of values defined continuously over a time interval. It is an abstraction on a continuously variable control output produced by a system.

Event flows report a happening or give a command at a discrete point in time and have no data content. An event flow is an abstraction on an interrupt or other "pulse" type message in a system.

Protocol data flows use a communication protocol for the transfer of data. Protocol data flows are introduced in this report. As to why and how they are used see the chapter on protocol data flows.

Flows of a particular type may converge or diverge to represent multiple sources and multiple destinations.

2.1.3 Stores

Stores are represented by parallel straight line segments.
A data store stores data that is subject to storage delay. When the data store is accessed by a data transformation the data is not consumed. It is an abstraction of any device that can store data; for example: a RAM, hard disk, etc.

An event store and a buffer are both represented by the same notation. You can tell them apart by the kind of flow that goes in or out of them. An event store can store an event or events until consumed. A buffer can store data before it is consumed. A buffer is an abstraction on a stack, queue or device that stores data until the data is accessed. To indicate that the buffer is a stack or queue, "stack" or "queue" is used as part of the name of the buffer.

2.2 Elements of the Ward/Mellor method

2.2.1 Context Schema

The context schema is a description of the boundary that separates the system from its environment.

![Figure 2.4: Context Schema](image)

The system appears as the only transformation in the context schema. The terminators, drawn as rectangles, represent sets of things with which the system interacts. Only flows that cross the system boundary appear on the diagram.

2.2.2 Event List

An external event has three characteristics:
- It occurs in the system's environment.
- It elicits a preplanned response from the system.
- It occurs at a specific point in time.

A list of all the external events is called the event list. The event list is useful to make a preliminary model of required system behaviour.
2.2.3 Transformation Schema

A transformation schema is made up of flows, stores, data transformations and control transformations. The transformation schema allows a description of the system that encompasses both data relationships and the behaviour of the system over time. The data and control connections among the transformations are shown. A data transformation may be decomposed into data transformations and control transformations; giving a transformation schema at a lower level.

2.2.4 State Transition Diagram for the Control Transformations

The logic of a control transformation can be described by a Mealy-type finite machine. The state diagram uses rectangular shapes rather than the more common circular shapes to represent states to avoid confusion with the symbol for a transformation.

A simple example will illustrate the state transition diagram. A machine is turned on or off by pushing a button. When the button is pushed the machine may go on or off according to what happened before. The control transformation is given in the following figure.

![State Transition Diagram](image)

Figure 2.5: Control transformation

Conditions cause the system to make a transition. Actions are taken as the transitions occur. The conditions are written above the line that separates the condition and action of a transition; the actions are written below the line.
Figure 2.6: State Transition Diagram

In the example push is a condition. MachineOn and MachineOff are actions. The state diagram has two states; off and on. The unconnected arrow that points to the off state indicates that this is the initial state.

2.2.5 Description of the Data Transformations

The data transformations are described in "structured English". Structured English uses a subset of English that is formal and restricted enough to be unambiguous to a reader but that has no precise syntactic requirements. It refers only to variable data elements or groups that are defined as contained in the flows and stores used in the transformation schema. It uses clear imperatives such as "find", "store", "select the largest", and so on, to refer to operations on data elements. It connects operations on data elements only with structured programming constructs (sequence, if-then-else, various closed loop constructions). It uses indentation and numbering to clarify the structure of the logic. The names of the flows and stores are capitalized.
2.2.6 Data Dictionary

The data dictionary lists all the flows and stores alphabetically. The data dictionary provides a description of the data contained in the flows and data stores of the transformation schema. The following notations are used for the composition of data:

<table>
<thead>
<tr>
<th>notation</th>
<th>read as</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>together with</td>
</tr>
<tr>
<td>{...}</td>
<td>select one of</td>
</tr>
<tr>
<td>{...}</td>
<td>iterations of</td>
</tr>
<tr>
<td>x(...)y</td>
<td>at least ( x ) but no more than ( y ) iterations</td>
</tr>
<tr>
<td>@</td>
<td>identifier for access of data in store</td>
</tr>
</tbody>
</table>

The meaning of the data item is set off by asterisks.
3 PROTOCOL DATA FLOWS

This chapter will describe the protocol data flows and explain why they are introduced to supplement the other types of flows.

3.1 Stores and their Dataflow

The following, regarding stores, is according to Ward/Mellor.[1]

Dataflow from transformation A to the store is interpreted as a modification of the content of the store. The store is changed in value or in set membership by transformation A. The flow from the store to transformation B is interpreted as a (nondestructive) use of the content of the store by transformation B.

Transformation B will use data that transformation A has produced. The store thus represents a time-delayed relationship between the transformations. Changes to, or uses of, a store are represented as occurring at discrete points in time, and thus the flows are discrete data flows.

Up to here is what Ward/Mellor described regarding stores.

The representation of a store with its dataflows can be misleading. Figure 3.1 suggests that the store is the one that actively gives the data to transformation B, while this is not so. Transformation B actually has to communicate with the store by means of a control flow or a discrete data flow to access the data that it needs from the store.
A more correct notation would be the following:

Figure 3.2: Two better store notations.

We can now see that the two discrete data flows of fig. 3.1 are actually very different. The notation of figure 3.2 is better but it forces the designer to consider the precise way in which the store is to be accessed. It also implies a more complex behaviour of the store. (Note that the notation of figure 3.2 is not used by Ward/Mellor nor will it be used in this report. The figure illustrates that there is a difference between the two flows.) This is a disadvantage at a high modelling level. In order to differentiate the two data flows the following notation is introduced:

Figure 3.3: New notation for a store

The protocol data flow indicates that the resulting discrete data flow was due to transformation B asking for it. How it is exactly accessed is not so important at a high modelling level.
3.2 Decomposition of Protocol Data Flows

In digital systems protocols are often used to transfer data. The protocol can, for example, be a request/acknowledge protocol. By experience with the modelling method it was found that there was a need for a flow that could model this.

![Protocol Data Flows](image)

**Figure 3.4: Protocol Data Flows**

Figure 3.4a denotes data that is sent from data transformation X to data transformation Y. Transformation X is the master and Y the slave. In this case the protocol data flow is comparable to the discrete data flow that Ward/Mellor use.

Figure 3.4b denotes that data is sent from data transformation X to Y where Y is the master and X the slave. Data transformation Y will have to request the data from transformation X.

In both cases correct transmission of data is assumed. There is no loss of data. Since the protocol data flow is a "high-level" flow it is possible to decompose it into flows at a lower level.

The protocol data flow can be modeled as being any protocol used for the exchange of data between two transformations.
The following figure will illustrate a decomposition of the protocol data flows according to increasing number of protocol signals:

A decomposition of the protocol data flow to a fully interlocked handshake protocol would result in a continuous data flow with four signals.

Note that the names of the protocol signals are made by adding a suffix to the name of the data that is sent. This keeps the naming of signals simple and shows which data the signals correspond to.
4 QUEUE

In this chapter a queue will be modeled according to the Ward/Mellor realtime method [2]. Different models of the queue will be made and evaluated. The difference consists of the way the environment (the source and the sink) behaves. The behaviour of the environment and thereby the queue depends on the chosen manner of data exchange to and from the queue.

In all the models the queue will be a passive device. The queue doesn't take the initiative.

In 4.1 the queue and its memory management are described. All the models of the queue use the presented memory management method.

In 4.2 a model of a queue is described which makes use of the fully interlocked handshake protocol. The manner in which the handshake protocol is used makes it accomplish two things. One of them is that the data is exchanged by the fully interlocked protocol. The other is that it ensures that data exchanges between the devices only take place when the queue is ready for the data. This means that no data is written when the queue is full or read when empty.

In the queue models presented in 4.3 a clear distinction will be made between the data exchange and the queue status. The introduced protocol data flow is used here. The simplest decomposition of the protocol data flow will be used to make the models.

In 4.4 an evaluation of the made models is made.

4.1 General Queue Description

A queue can be described as a buffer between a source system and a sink system. It processes its data first in first out (fifo). Given is that the queue is unidirectional. This means that it cannot use its input connections to give an output, nor use its output connections to receive an input. Data travels in one direction.

Using a queue between the source and the sink often has a positive effect on the time that the source system spends waiting for its data to be read.

The queue should:
- maintain fifo sequence of data
- accept no more data if it is full
- not be able to give data if it is empty.

The queue memory that is used to buffer the data can be represented as a continuous succession of memory locations where the first memory location
follows the last memory location. This is called wraparound.

Pointers are used to maintain the fifo sequence of data. The write pointer, Wp, points to the memory location to be written into. The read pointer, Rp, points to the location that is to be read from.

Figure 4.1: Queue memory with pointers

If a data word is written into the queue the Wp is moved to the following memory location. Likewise, if a data word is read from the queue the Rp will then point to the following memory location. This is how the pointers keep track of the data to be read or written. As long as the pointers point to different locations data can be read and written.

If the pointers point to the same place it either means that the queue memory is full or empty. (see fig. 4.2)

Figure 4.2: Queue memory full or empty
To be able to distinguish between the two cases the last operation on the queue has to be known. If the last operation was a write and the Wp and Rp point to the same place then the queue is full. A write is not allowed because it is full. If otherwise the last operation was a read then the queue is empty. Now a read is not allowed.

In the initial situation the queue is empty. Rp and Wp point to the first memory location. This means that the initial value for the last operation is a read.

We now have the following three cases for the allowed operations on the queue:

\[ Rp \neq Wp \]: data read and data write are allowed
\[ (Rp = Wp) \text{ and } (\text{last operation} = \text{read}) \]: The queue is empty. Data can only be written in the queue.
\[ (Rp = Wp) \text{ and } (\text{last operation} = \text{write}) \]: The queue is full. Data can only be read from the queue.

This description of the queue is used in the data transformations of all the queue models.

4.2 QUEUE with Fully Interlocked Handshake Protocol

The queue makes use of a fully interlocked handshake protocol. The handshaking ensures that data exchanges between the devices only take place when the devices can properly respond. It is called handshaking because after each action a reaction follows. Once the whole cycle is completed we always return to the beginning situation.

The handshake protocol between the source and the queue proceeds as follows:
- queue signals it is ready for data (IRFD)
- source signals that it has data (IDAV)
- queue will read in the data and once the data is "clocked in" will signal IACK
- source can remove data and gives IDNV. The source now will wait for an IRFD from the queue.

The sequence for the output is similar:
- the queue indicates it is ready by the signal Not Busy
- the sink signals it is ready for data (ORFD)
- queue signals that it has data (ODAV)
- sink will read in data and once it is "clocked in" will signal OACK
The queue will now remove the data and signal Not Busy, and once more we are in the beginning situation.
Note that the queue is modeled as a passive device. The handshake protocols at the source and sink side are not equivalent. A queue can not be connected after the other because neither would take the initiative.

Now the different elements that will constitute the total queue model will follow.

4.2.1 Context Schema

- DATA IN
- IDAV Input Data Available
- IDNV Input Data Not Valid
- IRFD Input Ready For Data
- IACK Input Acknowledge
- DATA OUT
- ODAV Output Data Available
- ORFD Output Ready For Data
- OACK Output Acknowledge

Figure 4.3: Queue Context Schema

4.2.2 Event List

The signals of a protocol also are part of the event list.

The event list for the queue is:
- IDAV
- IDNV
- IRFD
- IACK
- ODAV
- NOT BUSY
- ORFD
- OACK

This event list is useful for the construction of the data and control transformations.
4.2.3 Transformation Schema

Now that the handshake protocol and the queue memory management have been described a transformation schema of the queue can be made. The transformation schema of the queue (see fig. 4.4) consists of a control transformation, three data transformations and several data stores and event stores. The control transformation is called the Arbiter. It controls the behaviour of the data transformations by activating them.

4.2.4 State Transition Diagram for the Control Transformations

The Arbiter is described in a state transition diagram. (fig. 4.5) The state transition diagram consists of three main parts. They are for when the queue is empty, the queue is full and for when the queue is ok. In order not to increase the complexity of the state transition diagram the event store is used. The event store remembers that an event has occurred, so that an event can be processed later. Using an event from an event store entails removing the event from the event store.

The event store is used because of what can occur in the following situation. Looking at the Arbiter state diagram, when the Arbiter is in the READY IN/OUT state it can react to either IDAV or ORFD. If it receives the condition IDAV first it should be able to process the condition ORFD while going through the states PREPARE DATA IN and DONE IN. The same observation can be made of IDAV if ORFD is received first in the state READY IN/OUT. This can be taken care of by adding extra states to the state diagram or by using event stores. So as not to make the state diagram too complex an event store is used for IDAV and ORFD (see fig. 4.4).
Figure 4.4: Transformation Schema
Figure 4.5: State Diagram of the Arbiter
4.2.5 Description of the Data Transformations and Initialization

The data transformations of the transformation schema are Write Data, Read Data, and Determine Queue Status. They are described in "structured English" :

**Write Data**
if WRITE occurs then
begin
(1) Write one data word into the data buffer along with its WRITEPOINTER.
(2) if WRITEPOINTER <> MAX
    then WRITEPOINTER := WRITEPOINTER + 1
    else WRITEPOINTER := 1;
(3) LAST OPERATION := W;
(4) WRITE DONE;
end;

**Read Data**
if READ occurs then
begin
(1) Output the data word from the data buffer that corresponds
    to the current READPOINTER.
(2) if READPOINTER <> MAX
    then READPOINTER := READPOINTER + 1
    else READPOINTER := 1;
(3) LAST OPERATION := R;
(4) READ DONE;
end;

**Determine Queue Status**
if STATUS occurs then
begin
if ((READPOINTER = WRITEPOINTER) and (LASTOPERATION = R))
    then EMPTY signal given;
if ((READPOINTER = WRITEPOINTER) and (LASTOPERATION = W))
    then FULL signal given;
if READPOINTER <> WRITEPOINTER
    then OK signal given;
end;

Ward/Mellor do not describe where and how the data concerning the initialization of the stores should be given. These are the values of the stores on
initialization:

LASTOPERATION := R;
READPOINTER := 1;
WRITEPOINTER := 1;

4.2.6 Data Dictionary

MAX is the maximum amount of data words that the queue can store.

<table>
<thead>
<tr>
<th>Data</th>
<th>$1{\text{@address + binary number}}\text{MAX}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataIn</td>
<td><em>binary number</em></td>
</tr>
<tr>
<td>DataOut</td>
<td><em>binary number</em></td>
</tr>
<tr>
<td>IACK</td>
<td><em>Input Acknowledge</em></td>
</tr>
<tr>
<td>IDAV</td>
<td>*Input Data Available</td>
</tr>
<tr>
<td>IDNV</td>
<td><em>Input Data Not Valid</em></td>
</tr>
<tr>
<td>IRFD</td>
<td><em>Input Ready For Data</em></td>
</tr>
<tr>
<td>Last Operation</td>
<td>*values: [R</td>
</tr>
<tr>
<td>Not Busy</td>
<td>**</td>
</tr>
<tr>
<td>ODAV</td>
<td>*Output Data Available</td>
</tr>
<tr>
<td>ODNV</td>
<td><em>Output Data Not Valid</em></td>
</tr>
<tr>
<td>ORFD</td>
<td><em>Output Ready For Data</em></td>
</tr>
<tr>
<td>Read</td>
<td>**</td>
</tr>
<tr>
<td>Read Done</td>
<td>**</td>
</tr>
<tr>
<td>Readpointer</td>
<td><em>values: integer, 1..\text{MAX}</em></td>
</tr>
<tr>
<td>Write</td>
<td>**</td>
</tr>
<tr>
<td>Write Done</td>
<td>**</td>
</tr>
<tr>
<td>Writepointer</td>
<td><em>values: integer, 1..\text{MAX}</em></td>
</tr>
</tbody>
</table>
4.3 Queue Models with Protocol Data Flows

The queue will now be modeled with the introduced protocol flows. When protocol data flows are decomposed, the decomposition with the least amount of signals will be used. In these models the object is to get a clear distinction between the data exchange and the workings of the queue status.

4.3.1 Level 0 Queue Model

Using the protocol flow gives the following model at the highest level:

![Figure 4.6: Level 0 Context Schema](image1.png)

The above transformation schema makes use of the notation for a buffer [4]. This is a special type of store. A buffer is an abstraction on a stack or queue. A flow from the buffer is interpreted as removing a data unit from storage; in other words a destructive use of content. This store could be described as being intelligent because it "knows" what data to remove when accessed. When the buffer notation is used you have to specify if the buffer models a queue (fifo) or a stack (lifo).

Note that the notation for a buffer and an event store are the same. One can notice the difference between them by the kind of data that goes into them. A buffer uses discrete data flows and an event store uses control flows.

![Figure 4.7: Level 0 Transformation Schema](image2.png)
4.3.2 Queue in Polled Mode

When we model the queue at a lower level the queue itself will have to be modeled with a "dumb" store and the protocol data flows will have to be decomposed. The simplest decomposition of the protocol data flows give the following context schema:

![Context Schema](image)

This will not give a correct model because the source doesn't know if the queue is full and the sink doesn't know if the queue is empty. The source would be able to send data when the queue is full resulting in loss of data.

A status store is introduced to take care of this. Before sending data to the queue the terminators will have to check the status of the queue. The queue is thus accessed by polls from the terminators. The source and the sink poll the queue to see if they can respectively send or receive data. The status can either be full, empty or OK. Care has to be taken that the data is accessed correctly. Only one data transformation can access the store at a time. If this is not done nothing can be said of the correctness of the data.

![Context Schema Polled Mode](image)

The status protocol flows come from a common store. The flows that come
Figure 4.10: Transformation Schema of Queue in Polled Mode
from stores will not be decomposed as this is the notation that suits the flow coming out of a store the most.

Following these design ideas the transformation schema and the state diagram are made. (See figures 4.10 and 4.11.)

Note that two event stores are used in the transformation schema.

Figure 4.11: State Transition Diagram Polled Mode
The data transformation descriptions are as follows:

Write Data
if DATAIN occurs then
begin
(1) send WRITE REQ;
(2) When WRITE is received write DATAIN input into DATA along with its WRITEPOINTER.
(3) if WRITEPOINTER <> MAX
    then WRITEPOINTER := WRITEPOINTER + 1
    else WRITEPOINTER := 1;
(4) WRITE DONE;
end;

Read Data
if DATAOUT.REQ occurs then
begin
(1) send READ REQ;
(2) When READ is received output the data word from DATA that corresponds to the current READPOINTER.
(3) if READPOINTER <> MAX
    then READPOINTER := READPOINTER + 1
    else READPOINTER := 1;
(4) READ DONE;
end;

Determine Queue Status
if WRITE DONE occurs then
begin
if (READPOINTER = WRITEPOINTER)
    then STATUS := full;
if READPOINTER <> WRITEPOINTER
    then STATUS := ok;
end;

if READ DONE occurs then
begin
if (READPOINTER = WRITEPOINTER)
    then STATUS := empty;
if READPOINTER <> WRITEPOINTER
    then STATUS := ok;
end;
**Data Dictionary**

MAX is the maximum amount of data words that the queue can store.

Data = 1[@address + binary number] \( \text{MAX} \)

DataIn = "binary number"

DataOut = "binary number"

DataOut.req = **

Read = **

Read Done = **

Read Req = **

Readpointer = "values: integer,1..MAX"

Status = [full|empty|ok]

Write = **

Write Done = **

Write Req = **

Writepointer = "values: integer,1..MAX"
4.3.3 Queue used in event mode

In this last model of the queue, the queue will generate an event if the status of the queue changes. The terminators should read the status if they get a Status Change event.

Figure 4.12: Context Schema Event Mode

The transformation schema is almost the same as for the polled mode. The only difference is the addition of the Status Change event flows and a different data transformation. The data transformation Determine Change and Status now not only writes into the Status store but also reads from it. This data transformation uses the local data element LastStatus. (see fig. 4.13 on the next page)
Figure 4.13: Transformation Schema of Queue in Event Mode
The Arbiter works in exactly the same way as the Arbiter of the polled mode. The data transformations Write Data and Read Data are also the same as in the polled mode.

**Determine Queue Status**

if WRITE DONE occurs then
begin
(1) LastStatus := STATUS;
(2) if (READPOINTER = WRITEPOINTER)
then STATUS := full;
   if READPOINTER <> WRITEPOINTER
then STATUS := ok;
(3) if LastStatus <> STATUS
then give STATUS CHANGE;
end;

if READ DONE occurs then
begin
(1) LastStatus := STATUS;
(2) if (READPOINTER = WRITEPOINTER)
then STATUS := empty;
   if READPOINTER <> WRITEPOINTER
then STATUS := ok;
(3) if LastStatus <> STATUS
then give STATUS CHANGE;
end;

**Data Dictionary**

MAX is the maximum amount of data words that the queue can store.

<table>
<thead>
<tr>
<th>Data</th>
<th>1[@address + binary number]MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataIn</td>
<td><em>binary number</em></td>
</tr>
<tr>
<td>DataOut</td>
<td><em>binary number</em></td>
</tr>
<tr>
<td>DataOut.req</td>
<td>**</td>
</tr>
<tr>
<td>Read</td>
<td>**</td>
</tr>
<tr>
<td>Read Done</td>
<td>**</td>
</tr>
<tr>
<td>Read Req</td>
<td>**</td>
</tr>
<tr>
<td>Readpointer</td>
<td><em>values: integer,1..MAX</em></td>
</tr>
<tr>
<td>Status</td>
<td>[full</td>
</tr>
<tr>
<td>Status Change</td>
<td>**</td>
</tr>
<tr>
<td>Write</td>
<td>**</td>
</tr>
<tr>
<td>Write Done</td>
<td>**</td>
</tr>
<tr>
<td>Write Req</td>
<td>**</td>
</tr>
<tr>
<td>Writepointer</td>
<td><em>values: integer,1..MAX</em></td>
</tr>
</tbody>
</table>
4.4 Evaluation of the Queue Models

The value of using protocol flows is apparent when the context schemes are compared. When protocol flows are used to make a context schema, the context schema becomes a lot simpler. It is also clear as to which device is the master or slave.

In the model using the handshake protocol, the fully interlocked handshake protocol is used for several things. The fully interlocked handshake protocol is used in such a way that it takes care of the problems that occur when the queue is full or empty. It also takes care of the data in the store being correctly accessed. The handshake protocol is thus used for more than only to transfer the data correctly.

The design of the queue in the polled and event mode has the advantage that it is more straightforward. The protocol for the exchange of data is only for the correct exchange of data. It doesn't also take care of the correct access of the queue memory. This allows a clear division of functions between the data exchange and the access of memory, with as effect a clearer model.

The store DATA used in the models needs a controlled access of data. This was done in order to model the access of data in a queue controlled by the Arbiter. The other stores that are used do not need this while the used notation for the stores is the same. A notation that indicates this difference should be considered.
5 DEVICE INTERCONNECTION FOR A COMPLETE INTERFACE SYSTEM

Already existing designs of a queue, a Multi Functional Input Output Controller (MFIOC), and a Universal Asynchronous Receiver/Transmitter (UART) are used to make a complete system [2],[3],[5]. Their interconnections will be described with the help of the Ward/Mellor notation.

The MFIOC is a programmable protocol converter which has parallel input and output data. A UART is an interface device between a modem and a CPU. It converts parallel data into serial format for transmission and converts incoming serial data into parallel data on reception.

Only two of the six modes of the MFIOC are used. It is used in either the fully interlocked input mode or the fully interlocked output mode. This is due to the queue being unidirectional and using a fully interlocked handshake protocol for its input and output. Each MFIOC is initialized and set in the right mode by the CPU.

To have an idea of how the devices should be interconnected a context schema will be made that uses protocol data flows as the connecting data flows. In order to be able to make this schema we have to know which devices take the initiative in the transfer of data.

The queue is a slave when connected to any device. The queue that is used is the queue with the fully interlocked handshake protocol. The UART works in the interrupt mode; it initiates the data transfer. As for the MFIOC, when it is connected to the CPU, the MFIOC is the master. This gives the following context schema.

![Context Schema with Protocol Flows](image)

Figure 5.1: Context Schema with Protocol Flows

\[\text{from} = \text{fully interlocked output mode} \quad \text{fim} = \text{fully interlocked input mode}\]
The CPU and the modem are the terminators; the devices are all seen as transformations.

Now the possibility of the connections being made between the existing models will be examined.
- The queue and the MFIOC both have well defined fully interlocked handshake protocols with which they exchange data. Because of this their connection is no problem at all.
- The MFIOC was designed to be able to connect a device to a CPU. This makes the connection between the CPU and the MFIOC straightforward.
- Now the connection of the MFIOC to the UART. Both the MFIOC and the UART are designed with a connection with the CPU in mind. This is not the case the way they are interconnected now. This could be problematic.

The following protocol data flow between the MFIOC and the UART will have to be examined more closely.

![Diagram of the connection between MFIOC and UART](image)
Figure 5.2: Connection between MFIOC and the UART

In order to be able to connect the MFIOC to the UART transmitter a description is first given of the two devices. When connecting the MFIOC to the UART transmitter the MFIOC is used in the fully interlocked input mode. When connecting the MFIOC to the UART receiver the connection is analogous. The following two descriptions are from [5].

![Diagram of MFIOC fully interlocked input mode](image)
Figure 5.3: MFIOC fully interlocked input mode
The MFIOC processes data like this:
- When MFIOC gets data by means of handshaking it signals InAttention to indicate that an InReqData event can be given.
- When InReqData is received the MFIOC then sends the discrete dataflow InData completing the handshake protocol.

As for the DART transmitter this is the sequence in which it handles data:
- The Host sends DataToTransmit
- Transmitter transforms DataToTransmit into SerOut
- When SerOut data is sent it gives the TransmitterAttention signal; signalling that the UART is now ready for DataToTransmit.

When we try to connect the two devices we notice that direct connection between the MFIOC and the UART is not possible. The UART does not have an input for the InAttention signal from the MFIOC. TransmitterAttention cannot be connected to the InReqData directly because the MFIOC could be doing something else other than waiting for an InReqData. If the TransmitterAttention signal is given at the wrong time it would be lost. The solution is to connect them by means of an event store:

---

Figure 5.4: UART Transmitter

Figure 5.5: Connection MFIOC to UART with store
Note that the dataflows InData and DataToTransmit are the same. They come from the respective descriptions of the MFIOC and the UART; their only difference is their names. The event store remembers that a TransmitterAttention was given. InReqData is then signalled at the right time to the MFIOC.

A solution that would make use of InAttention uses a control transformation as follows:

![Connection MFIOC to UART with transformation](image)

Figure 5.6: Connection MFIOC to UART with transformation

The state transition diagram for the control transformation Interface is:

![State Transition Diagram](image)

Figure 5.7: State Transition Diagram
This solution using the control transformation suggests that it works in about the same way as the event store does. This is correct. The event store as well as the state control transformation have memories. The MFIOC in fact "looks" at the event store to see if a TransmitterAttention was given.

Both of the solutions involve adding another device in order to connect the MFIOC to the UART. They both are possible solutions but a much more elegant solution would be to design the UART so that it can be connected directly to the queue. This is what is done in the next chapter.
Figure 5.8: Total Interconnections
6 UART

The investigation of the device interconnections in the preceding chapter leads to the subject of this chapter. In this chapter a UART will be designed that can be connected directly to the queue; a MFIOC is not needed to connect the queue to the UART. The resulting model of the UART will be another example of the Ward/Mellor real-time method. The model is based on [5].

A UART can operate in either the interrupt mode or the polled mode. The functional configuration of the UART is programmed during system initialization. This is done by having the CPU send FORMAT data to the UART. The format data is composed of: word length, clock rate, parity used, number of stop bits.

In the interrupt mode the UART will generate an interrupt to indicate that it has received data or that it can transmit data. In this case the UART initiates the data transfer.

![Fig. 6.1 UART used in interrupt mode](image)

In the polled mode the host polls the UART whether it is ready to transmit data or if it has received data. This information is contained in the status data. Here the host initiates the data transfer.

![Fig. 6.2 UART used in polled mode](image)
The UART will be used in conjunction with other devices. The UART is connected to a queue for its incoming and outgoing data. Since the queue is a passive device it can not initiate a data transfer. Therefore the interrupt mode of the UART is used.

![Diagram of total system interconnections](image)

**Figure 6.3: Total system interconnections**

### 6.1 CONTEXT SCHEMA

Now that we know the mode the UART will be used in the context schema can be made. The CPU, queues and modem will be considered the terminators, since the device to be modeled is the UART. The UART and its terminators constitute the context schema.

![Diagram of UART context schema](image)

**Figure 6.4: UART Context Schema**
The queues make use of a fully interlocked handshake protocol. Because of this the protocol flows ReceivedData and DataToTransmit are decomposed. The protocol flows are decomposed into continuous dataflows and signals. Note that for the first part of the names of the signals capital letters of the protocol flows are used. The full name is too large for convenient use in the schemes.

![Diagram of Context Schema with decomposed protocol flows](image)

**Figure 6.5:** Context Schema with decomposed protocol flows

### 6.2 EVENT LIST

- DDT.ack
- DDT.dav
- DTT.dnv
- DTT.rfd
- RD.ack
- RD.dav
- RD.dnv
- RD.rfd

The SerClk is also an event but it is a special one. The modem side of the UART is not modeled completely so SerClk is only mentioned.
6.3 TRANSFORMATION SCHEMA

The first decomposition of the UART divides it into three data transformations: Receiver, Transmitter and Update Format.

Figure 6.6: Level 0 Transformation Schema
The Receiver and the Transmitter are further decomposed.

Figure 6.7: Receiver Transformation Schema

Figure 6.8: Transmitter Transformation Schema
6.4 STATE TRANSITION DIAGRAMS

State transition diagrams are made for the control transformations of the UART receiver and transmitter.

![State Transition Diagrams](image)

Figure 6.9 Receiver and Transmitter State Transition Diagrams

6.5 DATA TRANSFORMATIONS

The data transformations are described in "structured English":

**UpdateFormat**
if FORMAT occurs on initialization
then FORMAT DATA is updated with new FORMAT data

**SendReceivedData**
if SEND DATA occurs
then get data from BUFFER and output this as RECEIVEDDATA
SerToPar
if SERCLK and SERIN give valid data
    then begin
        (1) Convert SERIN to parallel data according to the
            information in the FORMAT store;
        (2) give the signal NEW DATA;
    end;

ParToSer
if TRANSMIT occurs
    then begin
        (1) Convert the parallel DATA TO TRANSMIT into SEROUT
            according to the information in FORMAT;
        (2) signal TRANSMIT DONE;
    end;

6.6 DATA DICTIONARY

DTT.dav       = *DataToTransmit. data available*
DTT.dnv       = *DataToTransmit. data not valid*
DTT.rfd       = *DataToTransmit. ready for data*
DTT.ack       = *DataToTransmit. acknowledge*
DataToTransmit = *parallel data*

Buffer         = *temporary store for the received data*
Format         = *UART initialization information*
FormatData     = *store for FORMAT data*
NewData        = *event store*

RD.dav         = *ReceivedData. data available*
RD.dnv         = *ReceivedData. data not valid*
RD.rfd         = *ReceivedData. ready for data*
RD.ack         = *ReceivedData. acknowledge*
ReceivedData   = *decoded parallel data*

SerClk         = *clock with which data to and from the modem
                 is synchronized*
SerIn          = *serial data from modem*
SerOut         = *serial data to modem*
7 SUGGESTIONS FOR FURTHER RESEARCH

Further research could be done to see if it is possible to verify the model by means of simulations. This would imply that, under others, the structured English description would have to be replaced by a more formal language. An object oriented language would be suitable.

Since digital systems are modeled, research into the use of standard notations for registers, RAM, stacks, queues and other normally used components should be looked into. If this is done care should be taken to limit the number of introduced components. The transformation schema could become unreadable with too many component notations.

8 CONCLUSIONS

The Ward/Mellor real-time method is a usable method to successfully model digital systems. The graphic modelling language is a notation which is useful to get a "feeling" of the system being modeled.

The method assumes that the data transformations are infinitely fast. Since the modeled digital systems start at a level which are closer to the implementation model than the essential model this has its drawbacks. This is why the Queue model does have signals which indicate when a data transformation is completed.

The introduction of protocol data flows has the effect that the context schema becomes simpler. At a glance you can tell which transformation has the initiative in relation to another transformation. When used with stores it indicates that the stores are passive elements which was not so clear with the original notation.
REFERENCES

"Structured Development for Real-Time Systems." (vol 1,2,3) 
Prentice Hall, Englewood Cliffs, New Jersey 1986

[2] Richter, R. 
"Structured Design of a Queue-chip with emphasis on its Testability" 
Master Thesis, Eindhoven University of Technology 
August 1989

[3] Stevens, M.P.J., Loon, M.P.M. van: 
"Een multifunctionele I/O-bouwsteen." 
EUT report 84-E-143, TU-Eindhoven 1984

"The Transformation Schema: An Extension of the Data Flow Diagram to 
Represent Control and Timing" 
IEEE Transactions On Software Engineering 
Vol.SE-12, No.2, pp. 198-210, February 1986

[5] Stevens, M.P.J., Budzelaar, F.P.M.: 
"IC-Ontwerp op Systeemniveau." 
Technische Universiteit Eindhoven 
Vakgroep Digitale Systemen

[6] Weinberg, V.: 
"Structured Analysis" 
Prentice Hall, Englewood Cliffs, New Jersey 1980

[7] Steward, D.V.: 
"Software Engineering with system analysis and design" 