9-21-2001

A formal architectural specification of the automatic validation of the Everglades National Park hydrology system

William S. Caldwell II
Florida International University

DOI: 10.25148/etd.FI14052537

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A FORMAL ARCHITECTURAL SPECIFICATION OF THE AUTOMATIC VALIDATION OF THE EVERGLADES NATIONAL PARK HYDROLOGY SYSTEM

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTERS OF SCIENCE

in

COMPUTER SCIENCE

by

William S. Caldwell II

2001

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by William S. Caldwell II, and entitled A Formal Architectural Specification of the Automatic Validation of the Everglades National Park Hydrology System, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Shu-Ching Chen

William Kraynek

Xudong He, Major Professor

Date of Defense: September 21, 2001

The thesis of William S. Caldwell II is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dean Douglas Wartzok
University Graduate School

Florida International University, 2001

DEDICATION

I dedicate this thesis to my parents who always believed I could do it as long as they lived and to my wife who has stuck with me through thick and thin. Without the moral support of the three individuals and the actual support of my wife, this thesis would never have reached fruition. Thank to all three of you. This document belongs, in a large part, to you.

ACKNOWLEDGMENTS

This research project was supported in part by the following agencies:

(1) The research team at The Everglades National Park (ENP) in southern Florida. I have been allowed to use their computer facilities and perform a case study on some of their software.

(2) It was also supported in part by the research team at The Biological Research Division of The United States Geological Survey (USGS–BRD). I have used their database (Data For Ever) and the system of support programs that manipulate it as a case study for this thesis. This database and supporting programs are implemented on the ENP computing system mentioned in (1) above.

(3) I have also been supported by the Southeastern Environmental Research Project (SERP). SERP is a research project that is headquartered here at Florida International University. SERP works in cooperation with ENP and USGS–BRD. I was originally hired by SERP to work as an assistant to the hydrology team at USGS–BRD. I was also allowed and encouraged to produce this thesis using their project as a subject and a case study.

(4) I have also been supported in part by The National Science Foundation . I am a member of the team that is operating the fellowship grant named "Formal Architectural Specifications a Design of Real-Time Distribution" and is governed by Dr He.

I wish to thank the members of my committee: Dr. Xudong He, Dr. William Kraynek and Dr Shu-Ching Chen. I would like to express special appreciation for the assistance I have received from my major professor, Dr. He, and for his help with the

setting up and implementing this project. I would also like to thank Dr. Kraynek and Dr. Chen for being members of my committee. Their assistance and suggestions and their scrutiny of my work have been invaluable. I would also like to thank Kevin Whalen and Krista Walker for their assistance in acting as an intermediary between the SERP and USGS–BRD. I would like to thank Gordon Anderson and Kevin Koten for their assistance with the interpretation of the existing software, Data For Ever. Finally I would like to thank my wife Elizabeth for her help and encouragement in completion of this project. Without her encouragement, assistance and proof-reading of this manuscript, it would not have been readable. I also acknowledge that this proof-reading is an extremely difficult process when she does not have an in-depth understanding of the subject matter, however the final result is a much more readable product because of her effort.

ABSTRACT OF THE THESIS

A FORMAL ARCHITECTURAL SPECIFICATION

OF

A CASE STUDY

AT

EVERGLADES NATIONAL PARK

by

William S. Caldwell II

Florida International University, 2001

Miami, Florida

Professor Xudong He, Major Professor

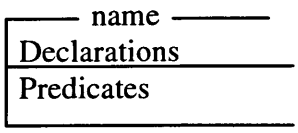
This thesis is focused on creating a formal architectural specification of a software system. I have used the computer system at the Everglades National Park (ENP) to conduct a case study. The formal method used is called Z ("zeta"). The case study used was performed on some of the software on the ENP computing system. The software is the Hydrology System and consists of a relational database called Data For Ever and several subsystems that support it. The Data For Ever system is a relational database that stores hydrology data from the Everglades. The supporting software tools are the programs and files that manipulate this hydrology data. The hydrology data were collected from the Everglades swamp. The data collected are water level, water salinity, water temperature, etc. The data were then checked and stored in the Data For Ever database. The system is up and running and in use every day (that is 365/366 days a

year) of the year. I added some software tools to it. There was little or no change to the existing database, however the database was redefined in Z specifications. This process is called Reverse Engineering. New modules to test and manipulate the data were defined. At present the data are not being tested or verified automatically as they are being collected from the field. The new modules test and/or verify the incoming data to examine the quality of them. The data are tested to see if they are within preset limits. If they are not, a possible error is recorded. These errors are presented during the daily report that is already in use.

TABLE OF CONTENTS

LIST OF FIGURES

Z operators

	Z schema
:	Is of type in Z
::=	Is of type in BNF
?	Input.
!	Output.
Δ	Is changed
Ξ	Is not changed

Logical operators

$\wedge$			.and.
$\vee$			.or.
$\neg$			Not
$\exists$			There exists
$\forall$			For all
$p \Rightarrow q$			If p then q

Set operators

$\in$			Is a member of
$\notin$			Is not a member of
$\cap$	$p \cap q$	p .and. q	Intersection
$\cup$	$p \cup q$	p .or. q	Union
|			Or
$\times$			Concatenation
| @#$%^&* |			The length of @#$%^&*

Arithmetic operators

=	Equal
$\neq$	Not equal

Notation

&&	Comment
* (in the first column)	Comment because this has been previously defined
@ (in the first column)	Comment because this is an example, not a constraint
# (in the first column)	Comment because this field is not used by us
b	Blank

LIST OF ACRONYMS

ASCII	American Standard Computing Information Interchange.
BNF	Backus Naur Form
DIP	Duel Inline Pin.
ENP	Everglades National Park.
EOF	End Of File.
LAN	Local Area Network.
NT	New Technology.
SERP	Southeast Environmental Research Project.
Shef	Standard Hydrologic Exchange Format.
Unix	Universal Internet Exchange
USGS-BRD	The Biological Research Division of The United States Geological Survey.
Z	Pronounced (zeta). The name of the formal method used in this thesis.

DEFINITIONS

Julian date	The numeric day of the year.	January 1 = 1. December 31 = 365/366.
Normal time	= 3:34 PM	= 34 minutes past three in the afternoon.
Military time	= 15:34	= 34 minutes past three in the afternoon.
Raw data time	= 1534	= 34 minutes past three in the afternoon.

Hydrology system All the water related data that is collected in the Everglades swamp

Raw Data The water related data as it is collected in the field in the Everglades swamp

Measurement A number that represents some attribute of the water

Label A number that describes where and when a measurement came from

Parse Move data from the raw data files to The Data For Ever database

Data For Ever A relational database that holds data from the Everglades

Measurement A number that represents some attribute of the water

Label A number that describes where and when a measurement came from

Descriptor A number that describes what has happened to a measurement

PERMISSION TO QUOTE

If anyone quotes the information found in this thesis,

either verbally, electronically reproduced, or written,

they shall give credit as to the source of the information.

Chapter 1

1. Introduction

This thesis is a case study of the use of Formal Methods to perform Computer Software Architecture (computer software **high level design**). It will be based on a case study of an actual system that is up and running and in every day use. The system is located at Everglades National Park (ENP) which is located in southern Florida. The system is a client server system with UNIX based networking and multi users that are NT based. This system was developed like most systems using **standard methods** (define, design, code, test, maintain, update, etc.) of software engineering development. It is a robust system and generally satisfies the requirements of the users. This system, like most systems that work satisfactorily, can always use a few changes, additions or deletions. As the system has needed these changes, additions, or deletions, they have been implemented using standard methods as mentioned previously. In this thesis, we will present some proposed additions using **formal methods** for the architecture. Formal methods produce the same result as standard methods but in a more sophisticated manner. With the use of formal methods, the developer can mathematically manipulate the parameters of the system as it is being developed. He/she can also examine the status and condition of the system as it is being developed.

1.1 Problem Statement

In the infancy of software development, there were very few methods of any kind for the

development of computer software. You just sat down and did it. If you were lucky and had a knack for that sort of thing, maybe you would end up with something that worked. Life was good for the programmer back then. There were few if any rules. If you wanted you could sketch out some sort of flow chart, but systems were simpler then and nobody had developed any kind of standard methods. Actually, at first, there were no systems. Even coding was unstructured. There was little or no structure to any language and the first languages had freedoms built in. You could jump in or out of anything (loops, blocks, programs, etc.) anywhere at any time. Life was good. No rules.

1.2 Some Relief to the Problem

Like all good things, the life of the artist sitting in the back room doing things that nobody understands or cares about has come to an end. Systems have become increasingly more complicated and the ability to produce a working system has become increasingly more difficult, even for the most astute guru. As time has passed, it has become advantageous to use more sophisticated methods, and conduct the process of software development in an increasingly organized manner. Languages have become more formal. There are more rigid constructs (loops, blocks, databases, etc.). Structured programming became the fad. Now the object oriented approach has become the "in" thing. Software Architecture (high level design) has improved. Instead of a few bubble charts or some flow charts, many extensive systems have been developed to track software from the time of definition to the time it is up and running and beyond into maintenance and enhancements.

Most of today's systems have been developed using some of these more modern methods, and the methods that define and track these systems are getting better every day. For example: The Unified Modeling Language has been developed and has nine (that is 9) different kinds of diagrams[7] that can be used to describe operation of a system. There are dozens of these software development tools on the market, and more being produced every day. Formal methods are one group of them. Formal methods are used to perform Software Architecture. Software Architecture includes the first step of Software Engineering which is High Level Design.

1.3 Solution Statement

There are few methods that are in common use today that are considered to be formal. Although most of the methods used today are good, they cannot be used to manipulate or describe the definition or design of the system by mathematical formulae. There are methods to show that once a hypothesis is established at the coding level, the coding can be tracked to prove that the desired result is obtained, but there are few at the design level. To produce this result during design, formal methods are being developed. This will allow mathematical tracking and control through the entire development process. Formal methods are still in their infancy but they are on the fast track. Hopefully we will be able to track the progress of the state of the software through the entire development cycle (definition, design, coding, test, maintenance, update,

test, maintenance, update,

test, ~~~).

Chapter 2

2. Software – an art to a science

When we began to make automatic machines programmable, software development was created as a new discipline. Like many other disciplines, software development began as an undefined discipline. It was more of an art form. If you were naturally good at it, that was fine; however, if it did not come naturally there was no way to learn the process. As time went on the occupation became known as computer programming and the discipline became known as computer science. You could give it a try, and if it was your thing, you could probably do well at it. At first, if you didn't have a natural knack for it, there was nowhere to go. There were no schools of computer science and not even many books. As time went on, software development became more understood, new methods were developed and schools became available to teach interested individuals how to develop software. Today there are many rules and regulations, even some laws, on how one should write software.

2.1. The art of writing software

Software development has not been around long, just a few decades (50 years at most). In the beginning software development was an art form and like most art forms nobody knew exactly how it was done. The artist just sat down and produced his/her masterpiece (or hopefully masterpiece). Nobody even knew if it was good. I guess it was all good

because it lead to something. How could you tell if you were producing a good product if you didn't have a bad one to compare it to? At that time we all worked in a vacuum and if our product did anything there were few critics qualified to judge us. In short, if you knew what the commands in the programming language did you could be a programmer. There were no methods (formal or informal) to understand.

.

2.2. The science of writing software

As time went on, we kept on writing and writing. During this writing frenzy, we would pause every now and then to jot down some sort of chart and graph to keep track of how we were doing and where the different parts of the program were kept. These charts and graphs became more organized until finally they could be put together to form a science. Now the art form is using the definition and design methods to implement software.

2.3 Methods of writing software

Over the last half century or so, methods of developing software have been getting progressively more formal. In the beginning, nothing was sacred. You could do anything you wanted and there were few people to answer to. Even with the development of new tools, many projects were completed with little or no design and only a sketchy definition. Most programmers didn't want to waste time writing a design when they could be writing code. Another problem was that a lot of programmers were not familiar with the process of reading a design or definition, let alone writing one. Most

users did not know what a computer was capable of so how could they ask for the best possible algorithm.

2.3.1. Original methods of writing software

In the fifties, there were few methods of definition or design. If someone wanted a computer program written, they would usually just tell the programmer what results they wanted, and he/she would start to write. The most you would expect is for the programmer to jot down a few notes. Nobody knew what a database was because we could put data anywhere. You couldn't design a database because there weren't any disks. Why waste valuable time designing a database when you could define a constant or variable anytime anywhere as you went. There was no problem with operating systems because there weren't any. Everything was done in a batch mode. We would run a program, stop the machine, clear the machine (run a small program to set everything to zero), load the next program to be run and run it. To develop a program required several steps of running these batch programs.

a. You first had to type the program on punch cards.
b. Clear the machine
c. Load the compiler (a deck of cards).
d Run the card deck containing the program through the reader (twice in earlier models) to compile the program.
e Correct any errors in the program deck and repeat steps b, c, and d until you got it right.

f. Get an object deck from the compiler.

g. Load the linker (another deck of cards).

h Load all the object decks.

i. If you got any linking errors, go back to step e.

j If you didn't get any errors, you got a loadable module from the linker (a deck of cards).

k. You could now run your program and see if it did what you wanted.

l. If it didn't, go back to step e.

m. If it did, you were a success.

Who could write a formal specification for a procedure like that. Nobody even thought of formal methods, they were too busy coding and running decks of cards through the card readers.

2.3.2. Improved methods of writing software

As time went on the linkers and loaders became better, but the big breakthrough was the system. The system was a program that ran all the time and kept track of all your program modules. Now it became advantageous to have some methods to your operations. The computers had disk drives and we could save our files indefinitely. To keep track of all these modules required some systematic cataloging process. Directories were formed, libraries were formed, and finally systems were developed.

2.3.3. Current methods of writing software

Today most software is developed using some kind of methods. These methods are usually quite informal but they tend to be more formal every day. In the case study that is undertaken in this thesis, the current software is described with informal methods. The database is described in a format called a schema, This is similar to the schema we use in our formal Z methods. The problem is that the current methods are not formal enough to define the software mathematically. If you cannot define your software in mathematical form, you cannot manipulate the software mathematically and you cannot track the progress of the software as it is being developed. This example does show, however, that real life is getting more formal (at least in this example).

2.3.4. Future methods of writing software

It is hard to predict the future but I can guess. I can imagine a processing method that covers the entire spectrum of computer software. I can envision a system where we will define the problem in logical arguments. We will then create a set of design specifications based on the definition specifications and the tools, both hardware and software. The code will then be developed based on the design specifications, and finally the completed program will be tested based on the definition specifications. There are bits and pieces of this process available today and more being developed every day, but nobody has put it all together in formal specifications.

2.3.5. Methods used in this case study

The formal methods that we are using in this thesis are known as Z (zeta) specifications. Z specifications are one of several methods of identifying a problem formally. To identify a problem formally, you must be able to specify it in mathematical notation without pictorial aids. You can have all the pictorial aids you want (like flowcharts, state-charts, dataflow diagrams, etc.) but they won't be part of the formal specifications. The Z specifications that we are using in this thesis are intended to represent what is traditionally known as a high level design.

Chapter 3

3. Purpose and Scope

3.1. Purpose

The purpose of this research paper is twofold. The first is to study and demonstrate the use of the latest state of the art in Software Architecture using Formal Methods and Z ("zeta") specifications. The second is to do this study in a practical application ("Data For Ever" database at the ENP) to demonstrate that formal methods are a valuable tool in the application of software architecture.

3.1.1 Z Specifications

Z specifications are one of many formal methods that are in development. Although there are no formal methods that have been around for any length of time, Z has been around longer than most. Z is a set-based model. With Z, everything is depicted as a set. With the application of set theory, first order logic, and graph theory we can use Z specifications to define and design a system that can be manipulated mathematically.

3.1.2. Practical Application

We have chosen to demonstrate these methods using a practical application because it is more straightforward to demonstrate that formal methods have a useful place in software architecture. It is more restrictive to present a practical application than to present a

hypothetical one. In a hypothetical situation it is easy to drift away from reality to make a point. It is also harder to show the actual real life advantages of formal methods.

3.2. Scope – A case study

The scope of the study will be limited to the software architecture (high level of design) of the project. At this time formal methods include only high level design, but someday it may well be a comprehensive system that covers the entire spectrum of software development. I will mention more about this in chapter 7 on future work with formal methods. For this thesis we will present a case study of the hydrology system used by ENP research group and USGS-BRD. The hydrology system has been implemented on the ENP computing system and has been running for several years.

3.2.1. Background Information

The problems to be examined are the special requirements needed for the use of The Everglades National Park (ENP) Computer System by the United States Geological Survey – Biological Research Division (USGS-BRD). This system is used to handle hydrology data that is collected from monitoring stations throughout the Everglades. The uses of this computer system by ENP and USGS-BRD are very similar, but there are minor differences. The USGS-BRD wants to extend and enhance the functions of the computing system. These enhancements will be implemented in conjunction with the ENP philosophy, so they may be used by ENP.

3.2.1.1. Political Setup

The USGS is a division of The Department of The Interior of the United States Government and ENP is a park in the Parks department of The Department of The Interior as shown in Figure 1 below.

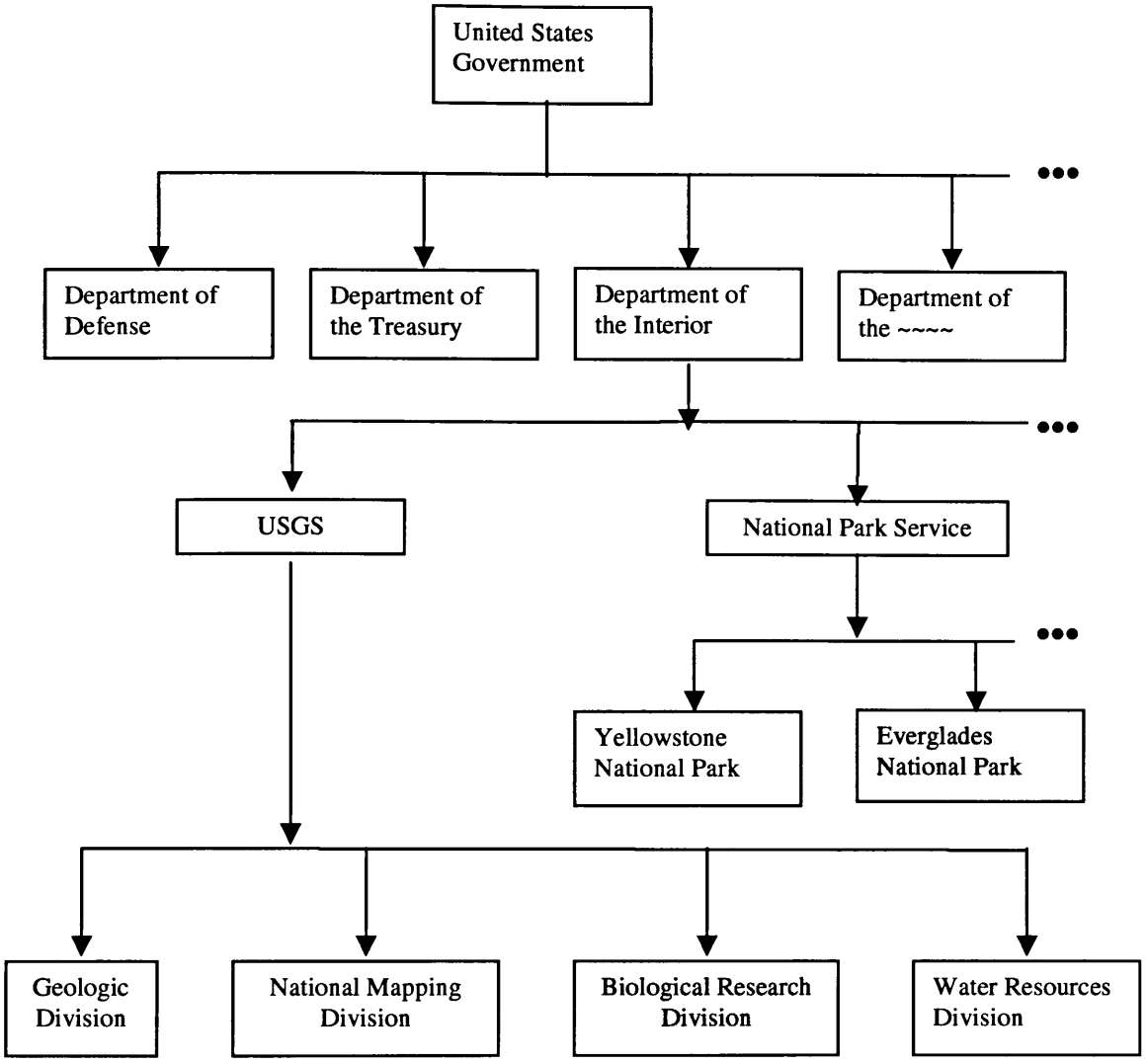


Figure 1
Structure of relevant part of the US Government

3.2.1.2. ENP Computing Setup

The ENP Computer System consists of two servers, both of which are connected together by a LAN. One server is a Novell based system with Windows NT workstations and the other server (named ALLIGATOR) is a UNIX based system with UNIX workstations. One of these UNIX workstations is called SNOOK and it handles the Hydrology Data for ENP, USGS-BRD, and other agencies. The hydrology data is data that comes in from

monitoring stations that are spread throughout the Everglades from Lake Okeechobee to Florida Bay. This data is collected via radio transmitters and modems, or cell phone and sent automatically by daily transmissions to a base station radio which is attached to a PC computer at ENP. There are about six hundred of these stations in the entire network. About one hundred of them belong to ENP and about sixteen of them belong to USGS-BRD. Most of the rest of the stations belong to The South Florida Water Management District. The general layout of the computer system at ENP and USGS-BRD is shown in the below in Figure 2.

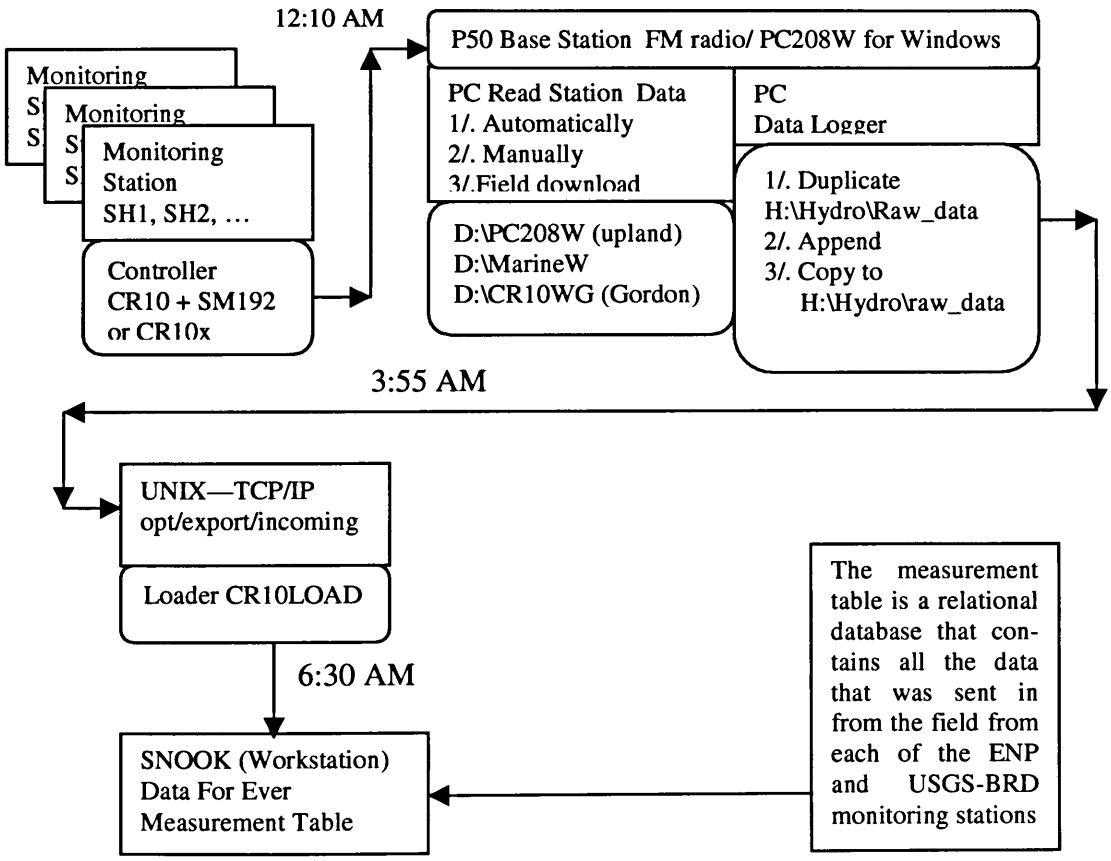


Figure 2.
Nightly data processing of ENP and BRD hydro data

3.2.2. The Problem Domain

The project has been implemented on the ENP Computer System in C++ and/or UNIX shell script. If the enhancements in this thesis are ever implemented they will be implemented on this ENP Computer System in either Java or UNIX shell script. The activities that need to be addressed for USGS-BRD Everglades Hydrologic Database are processing, validation and reporting of hydrology data

3.2.2.1. Processing of Hydrology Data

The several types of data that are collected go through several stages. If everything is working properly, the data is collected at each station for one day (24 hours) and accumulated in the CR10 storage unit that exists at each monitoring station. This data is called "Raw Data". Once a day, starting at about ten minutes after midnight, the data is sent to the base station via radio. At this point the data is still called "Raw Data". The data is arranged in files that hold one day's worth of information per file per station. The files are named xx.dat where xx is the name of the station. These files are stored in a directory called Raw_data. A printout of this directory is shown in figure 3. The data is then ***parsed*** into Data For Ever and is called "Intermediate Data". When the data is parsed it is rearranged so it is in a more logical order to be manipulated and examined. The several types of data that are taken and the steps they go through are as follows:

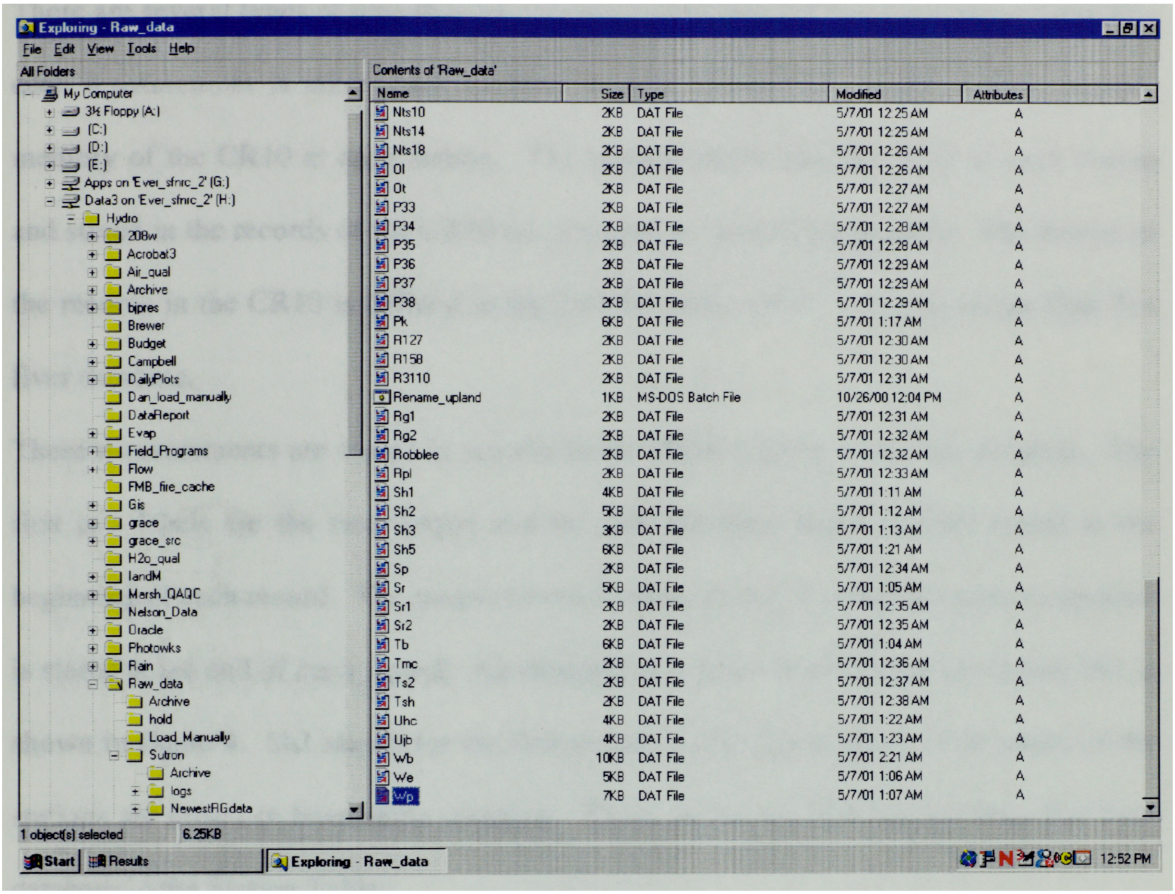


Figure 3.
Listing of the Raw_data directory

3.2.2.1.1. Types of data – labels and measurements

There are several types of data that are measured and collected at each station. Typically each measurement is taken once an hour at each station and stored in the computer memory of the CR10 at each station. The measurements that are taken at each station and stored in the records of the CR10 are arranged in chronological order. The format of the records in the CR10 is defined in the PARSE table, which is a table in the Data For Ever database.

These measurements are stored in records in the CR10's along with a set of labels. The first two labels are the record-type and the date-and-time stamp and are stored at the beginning of each record. The measurement(s) is/are stored next and the station signature is stored at the end of each record. An example of a days worth of data for station Sh1 is shown in figure 4. Sh1 stands for the first station on the Shark River. The names of the stations are meant to have some meaning. These names are listed in the Data For Ever database in the Station Table.

```
1,2001,129,129,1323.5,24.15,13.05,68
2,2001,129,130,-1441.7,19.81,13.05,68
6,2001,129,200,0,68
7,2001,129,200,.765,0,.766,.755,.723,.005,.735,.714,13.04,68
1,2001,129,229,1326.5,24.15,13.02,68
2,2001,129,230,-1443.5,19.75,13.02,68
6,2001,129,300,0,68
7,2001,129,300,.765,0,.766,.765,.721,.005,.735,.714,13.01,68
1,2001,129,329,1327.6,24.16,13,68
2,2001,129,330,-1448,19.61,13,68
6,2001,129,400,0,68
7,2001,129,400,.765,0,.765,.765,.721,.005,.735,.714,12.98,68
1,2001,129,429,1328.3,24.16,12.98,68
2,2001,129,430,-1452,19.49,12.98,68
6,2001,129,500,0,68
7,2001,129,500,.765,0,.765,.765,.724,.005,.735,.714,12.96,68
1,2001,129,529,1329.4,24.15,12.96,68
2,2001,129,530,-1454.5,19.41,12.96,68
6,2001,129,600,0,68
7,2001,129,600,.765,0,.765,.765,.726,.005,.735,.714,12.96,68
1,2001,129,629,1330.7,24.16,13.05,68
2,2001,129,630,-1457.5,19.32,13.07,68
6,2001,129,700,0,68
7,2001,129,700,.765,0,.765,.765,.726,.005,.735,.714,13.47,68
1,2001,129,729,1330.5,24.16,13.68,68
2,2001,129,730,-1448.6,19.59,13.69,68
6,2001,129,800,0,68
7,2001,129,800,.765,0,.765,.765,.721,.005,.735,.714,13.93,68
1,2001,129,829,1331.8,24.12,14.08,68
2,2001,129,830,-1436.2,19.98,14.08,68
6,2001,129,900,0,68
7,2001,129,900,.765,0,.767,.763,.722,.005,.735,.714,14.04,68
1,2001,129,929,1330.3,24.18,13.96,68
2,2001,129,930,-1411.5,20.77,13.95,68
6,2001,129,1000,0,68
7,2001,129,1000,.764,0,.767,.762,.723,.003,.726,.714,13.94,68
1,2001,129,1029,1331.4,24.16,13.88,68
2,2001,129,1030,-1389.5,21.49,13.87,68
6,2001,129,1100,0,68
7,2001,129,1100,.764,.001,.768,.761,.708,.007,.726,.694,13.84,68
1,2001,129,1129,1332.6,24.14,13.83,68
2,2001,129,1130,-1377.3,21.9,13.82,68
6,2001,129,1200,0,68
7,2001,129,1200,.765,.002,.768,.762,.706,.001,.706,.695,13.83,68
1,2001,129,1229,1332.4,24.16,13.81,68
2,2001,129,1230,-1373.2,22.04,13.81,68
6,2001,129,1300,0,68
7,2001,129,1300,.765,.001,.768,.763,.703,.004,.707,.696,13.85,68
1,2001,129,1329,1334.4,24.14,13.85,68
2,2001,129,1330,-1370.9,22.12,13.85,68
6,2001,129,1400,0,68
7,2001,129,1400,.766,0,.767,.763,.7,.005,.707,.681,13.85,68
1,2001,129,1429,1334,24.15,13.86,68
2,2001,129,1430,-1366.7,22.27,13.86,68
6,2001,129,1500,0,68
```

```
7,2001,129,1500,.767,0,.768,.764,.685,.002,.697,.676,13.88,68
1,2001,129,1529,1334.3,24.17,13.87,68
2,2001,129,1530,-1363.1,22.39,13.87,68
6,2001,129,1600,0,68
7,2001,129,1600,.768,0,.768,.765,.684,.003,.686,.676,13.91,68
1,2001,129,1629,1335.6,24.15,13.91,68
2,2001,129,1630,-1368.8,22.19,13.92,68
6,2001,129,1700,0,68
7,2001,129,1700,.767,0,.768,.764,.684,.003,.686,.675,13.93,68
1,2001,129,1729,1334.8,24.18,13.99,68
2,2001,129,1730,-1381.9,21.74,13.98,68
6,2001,129,1800,0,68
7,2001,129,1800,.766,.001,.769,.763,.668,.006,.686,.659,14.02,68
1,2001,129,1829,1335.6,24.18,13.74,68
2,2001,129,1830,-1393.6,21.35,13.64,68
6,2001,129,1900,0,68
7,2001,129,1900,.766,.001,.768,.763,.664,0,.664,.658,13.36,68
1,2001,129,1929,1336.1,24.16,13.33,68
2,2001,129,1930,-1409.6,20.83,13.32,68
6,2001,129,2000,0,68
7,2001,129,2000,.766,.001,.769,.763,.663,.001,.664,.653,13.3,68
1,2001,129,2029,1334.4,24.19,13.27,68
2,2001,129,2030,-1427.8,20.24,13.28,68
6,2001,129,2100,0,68
7,2001,129,2100,.765,0,.768,.763,.661,.003,.663,.652,13.26,68
1,2001,129,2129,1334.4,24.16,13.23,68
2,2001,129,2130,-1439,19.89,13.23,68
6,2001,129,2200,0,68
7,2001,129,2200,.766,0,.768,.763,.659,.004,.662,.651,13.21,68
1,2001,129,2229,1334.2,24.2,13.2,68
2,2001,129,2230,-1446.1,19.67,13.19,68
6,2001,129,2300,0,68
7,2001,129,2300,.766,.001,.767,.764,.658,.004,.666,.65,13.18,68
1,2001,129,2329,1335.5,24.15,13.15,68
2,2001,129,2330,-1453.1,19.46,13.15,68
99,68,4217,4.34
6,2001,129,2400,0,68
7,2001,129,2400,.766,0,.766,.765,.655,.004,.672,.65,13.13,68
1,2001,130,29,1335.9,24.16,13.1,68
2,2001,130,30,-1459.7,19.26,13.1,68
6,2001,130,100,0,68
7,2001,130,100,.765,0,.765,.765,.656,.005,.672,.65,13.08,68
```

Figure 4.
Listing of the Raw_data file for a monitoring station for one day.

Notes:

1/. The name of the station is Sh1 for the first station in the Shark River.

2/. The date is Julian 129 which is 05-09-01, which is 31 days for January plus 28 days for February plus 31 days for March plus 30 days for April plus 9 days for May.

3/. The last set of measurements is on the next day.

4/. The station signature is 68.

The collection of the day's measurements is radioed (or telephoned) in once a day, just after midnight. Not every type of data is collected at every station Several of the most common types data that are collected are:

3.2.2.1.1.1. Record Type (label)

The record type is a single number that is assigned to each record in the raw data file. Each time any measurement(s) is/are put into the raw data file, it/they is/are put in the record in a predefined order. This order is defined in the parse table in the Data For Ever database.

3.2.2.1.1.2. Station Signature (label)

The station signature is a unique number that is assigned to each station. Each time any measurement(s) is/are put into the database, it/they is/are put in the Measurement table (see Figure 8) and the station signature is put with it/them.

3.2.2.1.1.3. Date and time stamp (label)

The date and time stamp is basically just what it sounds like. They are the date and time that one or more measurement(s) were taken. Each date label consists of two numbers, the year and the Julian date. Each time consists of a single number and is the time in military format except there is no colon. These date and time stamps are put in the Data For Ever database in the Measurement table (see Figure 8).

3.2.2.1.1.4. Battery Voltage (measurement)

The battery voltage is the voltage of the battery and solar panel combined. Nearly every station reports the battery voltage every time any other measurements are taken. The purpose of this measurement is merely to check the condition of the station. At night the solar panel does not work and the voltage is expected to drop. There are a lot of reasons why the battery voltage can go too low such as, too much current drain, branches blocking the solar panel, the battery running out of water, etc. We cannot tell what the problem is, we can just report that it looks too low. The battery can also be too high and about the only thing that can cause this is that the voltage regulator has gone bad. Again we do not try to determine the cause we just report that the voltage is too high. Each time any measurement is taken a voltage reading (a single number representing the voltage of the battery and solar panel combined) is taken and put (parsed) in the database, in the Measurement table (see Figure 8) along with the measurement value, the station signature, and date and time stamp.

3.2.2.1.1.5. Rain (measurement)

Rain is simply the number of inches of rain that has fallen at each station since the last measurement (usually one hour). This parameter will be hard to check because here in South Florida we have local rain squalls that produce a lot of rain over a small area. If, on the other hand, a station is reporting rain when there was no rain in sight, we can flag a possible error. If we know that there has been rain everywhere, and no rain is reported, we can flag a possible error. Each time rain (a single number representing the number of inches of rain) is measured it is put (parsed) in the database, in the Measurement table

(see Figure 8) as the measurement value and the station and date and time stamp is put with it.

3.2.2.1.1.6. Water Level (measurement)

There can be three types of water at each station: 1/. surface water (water on top of the ground), 2/. soil water (water in the dirt), and 3/. groundwater (water in the bedrock). Each type of water has it's own water level. Each time the water level (a single number representing the number of feet above or below a fixed datum) is measured it is put in the database, it is put in the Measurement table (see Figure 8) as the measurement value (parsed) and the station and date and time stamp is put with it.

3.2.2.1.1.7. Salinity (specific conductivity) (measurement)

The salinity can be measured for each of the three type of water at each station. Each time the salinity (a single number representing the amount of salt in the water) is measured it is put in (parsed) the database, in the Measurement table (see Figure 8) as the measurement value and the station and date and time stamp is put with it.

3.2.2.1.1.8. Temperature (measurement)

The temperature can be measured for each of the three types of water, the outside air or the machinery inside the monitoring station at each station. Each time the temperature (a single number representing the temperature of the water or air)is measured it it is put in (parsed) the database, in the Measurement table (see Figure 8) as the measurement value and the station and date and time stamp is put with it.

Measurement Table
Example: The battery voltage at monitoring station Sh1

station	datatype	measurment_date	measurment_time	
Sh1	Voltage	05-09-2001	0129	
Sh1	Voltage	05-09-2001	0130	
Sh1	Voltage	05-09-2001	0200	

measurment_value	estimation_method	reason_value_missing	last_validation_date
13.05000			
13.05000			
13.04000			

last_person_validating	last_validation_process

Figure 5.
Data For Ever Measurement Table

3.2.2.1.4. Archival Data

Archival data is data that has been verified and corrected and has been in the system for a while. It is presumed to be in as good a condition as it ever will be. There are two types of corrections that have been made or can be made: 1/. Additions to fill in missing data and 2/. Corrections of bad data. We know that this data is not perfect and perhaps some advanced validation will be helpful.

3.2.2.2. Validation of Hydrology Data

As the data passes through the system, it can be examined and validated at any point. There are some points that are more logical to examine some data. There are some points that some data cannot be examined. At this point in time there is no examination of the data until the data reaches the Data For Ever Database. This means that if there is an error in any of the labels, there will never be any possible way to catch these errors. Referring back to section 3.2.2.1.1.1/2/3, the three labels are record type, station signature, and date and time stamp.

The record type is difficult to check. If it is wrong and the format of the record reported is the same as the format of true record, there will never be any way to tell. We may be able to guess if all the parameters are out of range, but nobody will ever know for sure.

The station signature can be checked because the stations are polled to send in their information by the base station. This means that the station signature can be

checked while it is being transferred from the monitoring stations to the base station. It can also be checked while it is in the xx.dat files or while it is being parsed into the Data for Ever database because the station signature should match the name of the xx.dat file. The station signature cannot be checked once the data is in the Data For Ever system.

The date and time stamp can also be checked while the data is raw data, but once the data reaches the Data For Ever database, there is no way to tell if it is correct. If some of the data is missing, it is impossible to tell if the hour is correct.

The measurements can be checked anytime but only for probable errors within reasonable limits. This is true for manual or automatic validation. The labels can be checked for exactness, but the measurements are always iffy.

When the data is collected from the field, it is called "Raw Data". This raw data may have a myriad of problems. The stations that collect the data are very complex and are composed of the latest state of the art equipment. While most of this equipment works well most of the time, sometimes it doesn't. It can go bad on its own or it can be invaded by a host of insects, reptiles, or small animals that live in the swamp. It can also succumb to the natural elements such as wind, rain, hurricanes, vegetation, lightning etc. These stations are also run by 12 volt battery power that is maintained by solar electric panels. At present there are twelve different problems that may occur with the system. They are stored in the Data For Ever database in a table called The Reason Value Missing Table. See figure 5a or 5b for the structure of the Data For Ever Database and the Reason Value Missing Table. The reasons that are currently recognized are:

1. Accumulated Later

Accumulated later is used on stations that collect rain data (the number of inches of rain that has fallen in the last hour) Some of these stations have an extra rain gauge that is the old fashioned type that does not use any electronic equipment. It is just a tube that holds up to ten inches of water. If the electronic computerized rain measurer quits, this simpler old fashioned device is quite likely to still work fine. The problem is that it does not record hourly rain amounts. It only records total amounts from the last time it was emptied, and it can only be emptied by hand and by a person visiting the site.

2. Device Malfunction

   This could be any of the pieces of equipment that collect, store, and send the data.

3. Power Failure

   This could be that the battery quit working because it went dry or it could be because the solar panel quit working, the voltage regulator quit working or any of the wires connecting these parts together became corroded or broken.

4. Sensor Out Of Range

   The sensors have to be calibrated against a known datum in order for the readings from them to be meaningful. If the known value is lost or the internal electrical properties of the sensor change, the reading of the sensor will be meaningless. If this error is very large, the sensor value will be out of range. If, on the other hand, the error is small, the reading will be just a little off and will be very hard to detect. If this latter case occurs there is no way to check for this kind of error except during periodic maintenance checks. This is the present case and with the implementation of these verification tools the situation will not improve much. In the future we may be

able to use a larger data bank and a more sophisticated statistical model to obtain better results. For now we will just rely on a high and low threshold to check for values out of range.

5. Sensor Out Of Water

This problem produces a result that is similar to the previous problem. The main difference is that the error is almost always very large and experience tells the hydrologist when this is occurring as opposed to Sensor Out Of Range.

6. Short Cable Hang-up

   Some sensors, especially water level, have a float on the end of a cable. The cable runs over a pulley and has a weight on the other end. These cables are forever getting tangled or hung up by plants or animals.

7. Site Maintenance

   Sometimes when a monitoring station has not been visited for a while, it just needs a good cleaning and adjustment.

8. Tip Bucket Plugged

   The tipping bucket is a mechanical bucket that tips every time there is 0.01 of an inch of rain. All sorts of foreign material can get in this mechanism and stop it

9. Unknown

   If all else fails we have this category.

10. Vegetative Influence

    There are a large variety of plants that grow in the everglades and no matter how hard you try, they always seem to be able to get in and clog up the works.

11. Well Went Dry

    Each monitoring station can have one or more wells to measure the subsurface water. If the well was not deep enough or the water level went below the minimum level, the well will go dry.

12. Wildlife Influence

    Like vegetation, there are a host of animals that live in the everglades and no matter how hard you try, they always seem to be able to get in. Sometimes they build a nest and clog up the works and sometimes they bite something and it doesn't work any more.

At any rate we do not have a continuous stream of data and so some of the data is not good. Some of the other software problems include:

1. Loss of clock or other values in the data logger (cr10).
2. Out of range values → 999 or 6999 flags due to system problems.
3. Data storage overrun, especially at the remote sights. This can easily happen when we have a site without a radio or a site where the radio quit working and we haven't gotten to fix it in time

If you refer back to figure 3 you will see that there is a Sh1, Sh2, Sh3, and Sh51 but no Sh4. For some reason monitoring station Sh4 is not reporting anything and at this point we do not know why. This is a good example of why all the data has to be validated to insure it's quality. At present, there is no automatic validation (automatic computer tools) and every bit of data has to be examined by hand. This produced the desire to

create a set of automatic validation tools that would enhance the validation process. The data validation process can be divided into several categories as follows:

3.2.2.2.1. Throughput Data Validation

Throughput data is the data that is automatically collected hourly, stored in the CR10 at each monitoring station for up to a day, transferred automatically to the base station early each morning, and transferred to The Data For Ever Database later each morning. At present there is no automatic mechanism to check for even the most horrendous errors

3.2.2.2.2. Batch Data Validation

Batch data is the data that is collected manually from time to time. Sometimes the stations work at partial capability. There may be data available at the station that was not automatically transmitted, for one reason or another. In this case, the data has to be collected by hand and then inserted into The Data For Ever Database by hand. At present this data also has to be validated by hand. It could probably be checked with the same set of tools that will be used on the through-put data.

3.2.2.2.3. Archival Data Validation

Archival data is old data that has been in the system for a while. This data has never been scanned by automatic validation tools because these tools didn't exist. These tools are currently being developed in this project.

3.2.2.3. Reporting of Hydrology Data

Whenever any of the new validation tools are run, they will produce a report. This report will be a list of possible errors. The computer will probably never be smart enough to determine if these errors are real or not. There will always have to be a hydrologist to look at the data to actually validate it. These tools merely make the job simpler. Reporting can be performed on a time interval basis without the intervention of an operator or it can be performed at any time on any group of suspicious data. For this thesis we will take the first step and get the validation programs working on a single station for a single time frame.

3.2.2.3.1. Daily Status Reporting

At present there is a daily report produced every morning automatically. The daily report is run automatically each morning after the data has been transferred to "Data For Ever". It shows the last week's data graphically as shown in figure 6. At present, no changes are made to the data. Even with new automatic verification tools there will probably be no changes to the data until a hydrologist has had a chance to examine it. After a hydrologist has examined the daily reports, he/she can run the daily status report again and ask the program to change or not to change the data for each or all cases that were reported as possible errors.

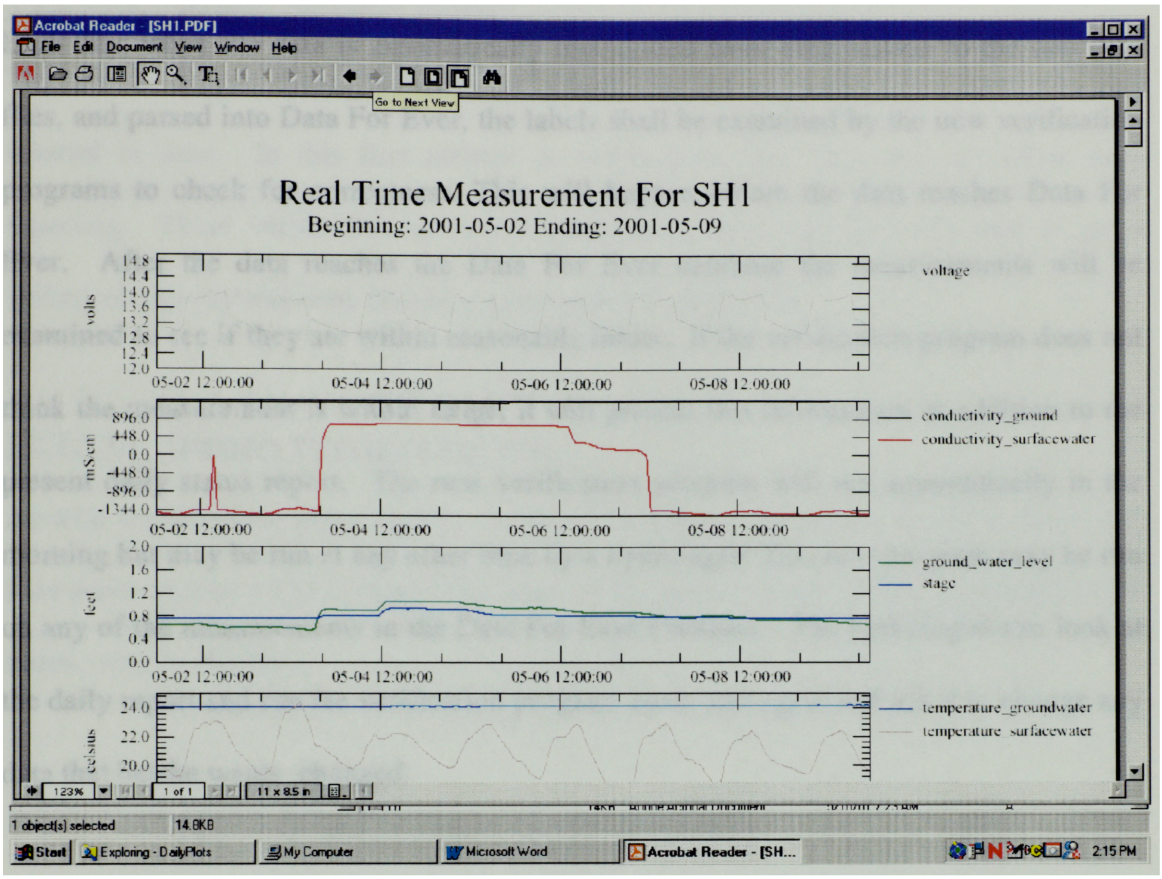


Figure 6.
Daily Report

3.2.2.3.1.1. Diagnostic Verification Aids

Each day, when the data is automatically transmitted from each station to the raw data files, and parsed into Data For Ever, the labels shall be examined by the new verification programs to check for correctness. This will happen before the data reaches Data For Ever. After the data reaches the Data For Ever database the measurements will be examined to see if they are within reasonable limits. If the verification program does not think the measurement is within range, it will present this information in addition to the present daily status report. The new verification program will run automatically in the morning but may be run at any other time by a hydrologist This new program may be run on any of the measurements in the Data For Ever Database. The hydrologist can look at the daily report and run the verification program again and again and ask it to change any data that he/she wants changed.

3.2.2.3.1.2. Informational

If any data is missing or null, this will be reported in the daily status report and the verification program can add the data. If the data is bad the verification program will be able to correct the data with the values supplied by the program itself or values supplied by the hydrologist.

3.2.2.3.2. Time-Series Reporting

At present the only report that is issued is a daily report. When the verification program is completed, it will run at the same time and in addition to the daily report.

3.2.2.3.2.1. Weekly, Monthly, Annual

This whole system leaves the door wide open for allowing summary reports at any interval in time. In this first attempt at verification aids, time will not allow such reporting. These verification measures should have been run for some time to get a feeling of security and trust that only running time can provide.

3.2.2.3.2.2. Irregular time-series reporting

As with regular time series reports, irregular (hurricane, full moon, drought, flood, etc) time series reports will be beyond the scope of this report. More will be mentioned in the future work in chapter 7.

Chapter 4

4. Methodology (Formal Methods using Z specifications)

The methods used in the high level design of this study are known as "Formal Architectural Methods". The formal methods that will be used in this study are known as "Z" ("zeta"). The first step will be to perform ***reverse engineering*** on the existing databases, the Raw Data database and the Data For Ever database. That is, to define the existing database in Z specification. The second step will be ***forward engineering*** to define the new functions (using Z specifications) that are to be added to the system.. The third step will be to implement the additions, in C^{++}, Java, and/or Unix shell. The fourth step will be to test the application that has just been built, and the final step is to demonstrate the project. The third, and fourth steps will be optional and they will be performed separately from this thesis.

4.1. Z (zeta) Specifications

Z notation is a language for expressing high-level design specifications of computer software systems. It is based on set theory and the notion that everything is a set. Z is also an object oriented language in which every ***thing*** is an object and every ***thing*** has a datatype. Therefore, every ***thing*** in Z is an object and a set and has a datatype. Z notation is presented in the form of a Schema. A Z schema consists of two collections or

sets. First is a collection of declarations, which is a collection of named objects and their datatypes. Second is a list of predicates. Schema's can define a static representation of the system, such as a database, or as an operational member of the system that produces a change in the system.

4.1.1. Z declarations

Declarations are statements that define the several objects of the system. Declarations in Z are similar to declarations in any other language. We will define each piece of data in a manner in which we can use them as operands in logical arguments. We then define each object that is composed of more than one data element. All declarations are in the form:

<variable name> : <DATATYPE>

The standard types of data are CHAR, INTEGER, DATE, and FLOAT. It is assumed that the reader is familiar with these types. All of the other types are made up of a combination of these basic types. The several types of data that are used in the system are listed in the Datatype Table in the Data For Ever system. We will not use all of them so I will list the ones that we are going to use. This list follows the Z schema of the Datatype Table presented in figure 16. The list is an example of what the Datatype Table could look like.

4.1.2. Z predicates

Predicates are statements that define the several operations that are performed on the objects of the system. They can be pre-conditions or post-conditions. A pre-condition

represents the condition of the system before the operation of the schema is performed. A post-condition represents the condition of the system after the operation of the schema has been performed. A predicate can also be an invariant, which states a relationship that is always true.

4.2. Reverse Engineering on the existing database

Reverse engineering is performed on the existing system software so the future development will work in harmony with the existing software. The modules that exist are included in a set of files named xx.dat (where xx is the name of the monitoring station) called Raw Data, a relational database called "Data For Ever", and several program files that manipulate the data in these files. Not all of the data files will have to be defined with Z specifications. Only the files that we will access will need to be specified in Z notation. This will allow the additions to the project to coincide with the existing software. The existing program files that store the programs that presently manipulate the data will not be considered in this thesis, therefore they will not have to be defined in Z specifications.

4.2.1. Databases

The two databases (the xx.dat files and the Data For Ever database) both contain the same data. An example of the two databases is shown in figures 4 and 5 above. There are, however, two differences in the two versions.

The first difference is in the size and content. The xx.dat files contain one day's worth (the previous day) of this data. The Data for Ever database contains all the data for all the stations for all time since this data has been collected (about five years).

The second difference is in the arrangement of the data. The data in the xx.dat files is arranged in chronological order. This is the order in which it was taken and stored in the storage module (CR10) in each station. Each hour each type of measurement (battery voltage, water temperature, salinity, etc.) is stored. When the data is moved from the xx.dat files to Data For Ever, the process is called **Parsing.** When the data is parsed into Data For Ever, it is stored in such a fashion that it can be looked at from several standpoints such as battery voltage, water temperature, salinity, etc. and it can be examined for any time period.

The two keys that link the two databases together are the Parse Table and the Station Table, both of which are part of the Data For Ever system. In the Parse Table is the definition of the data items that are in the xx.dat file. In the Station Table is the mapping between the station name and station number. This information has been inserted into the Data For Ever Database system via the Parse Table and the Station Table. It has also been coded into the firmware of each monitoring station. Hopefully the coding in the monitoring stations matches the coding in Data For Ever, but sometimes it doesn't. These are some of the errors that we are looking for.

4.2.1.1. Raw Data

The xx.dat files contain the raw data for one day that has just been received from each station. It is in ASCII format and is a string of numbers divided by commas with no

imbedded blanks. The first number is always the record type. The second number is the year and is four characters long. The third number is the Julian day of the year. The fourth number is the time in military format except there is no colon (:) and there is no leading zero on the hours before 1000. The remainder of the numbers are of varying length and divided by commas. The data type and position of these numbers is defined by Parse Table in the Data For Ever database. The last number is the station ID. An example of a raw data file is shown in figure 4 above.

4.2.1.2. Data For Ever

Data For Ever holds all the data that has ever been received from anywhere in the Everglades at any time (since this system was installed). Data For Ever is divided into about 20 files, each of which holds a table in the Data For Ever system. We will not address all of the tables. Only the tables that we use will be analyzed and described in formal methods. The tables that we are using are the Measurement Table, the Station Table, the Parse Table, the Station Datatype Table, the Datatype Table, the Manual Validation Event Table, the Person Table, the Validation Process Table, the Estimation Method Table and the Reason Value Missing Table.

4.2.1.2.1. Data For Ever Measurement Table

The focal point in the Data For Ever database is the Measurement Table. The Measurement Table holds all the measurements that were ever taken in the history of hydrology measurements in the Everglades (about 5 years). Each record in the

Measurement Table represents a single measurement from the raw data file (remember that a record in the raw data file could have several measurements in it). Along with each measurement, in the measurement file, are all the supporting ***labels*** (station, datatype, measurement time and measurement date) to tell what the measurement is all about. Each measurement also has a set of ***descriptors*** (estimation method, reason value missing, last validation date, last person validating and last validation process) that tell about the operations that have taken place on the measurement.

4.2.1.2.2. Data For Ever Station Table

The hydrology system consists, in part, of several monitoring stations. These stations take periodic measurements of the condition of the water in the swamp. The measurements are usually taken every hour, but this is not a hard and fast rule. The measurements that are taken are conditions such as temperature, salinity, water level, etc. Each of these stations is a little different than any other. This means that we have to describe each station so we can tell how to operate the hydrology system a little differently for each station. The Station Table is used for this purpose. We will only use the first two records of the Station Table, so I will only define the first part of the Station Table in formal specifications. The first two records are the station name and the station number. This is a one to one mapping, that is each station has a unique name and a unique number. There is one record for each station and each record contains a station's name and number. This name and number is used as ***part of the key*** to map the location of the data in the Raw Data database to the location of the data in the Data For Ever database. The remainder of the key is in the Parse Table, which is described next.

4.2.1.2.3. Data For Ever Parse Table

The Parse Table is ***the remainder of the key*** that maps the location of the data in the Raw Data database to the location of the data in the Data For Ever database. The first part of the key is in the Station Table, which was just described above. The Parse Table contains the location of each type of data in the Raw Data database.

4.2.1.2.4. Data For Ever Station Datatype (of each measurement) Table

The Station Datatype Table defines the datatype of each measurement at each station. There is one measurement for each type of measurement that is taken at each station. If there are about 600 stations in the entire system and each station has about 10 different measurements, then there should be about 6,000 entries in the Station Datatype Table.

4.2.1.2.5. Data For Ever Datatype Table

Every object in any system, including this one, has a datatype. In Object Oriented programming the developer can create his/her own datatypes if the basic types are not fancy enough to suit them. These datatypes are listed in the Datatype Table. This is a listing of the names of the datatypes, not a description of the datatype. For example, you may have two datatypes STATION_NAME and STATION_NUMBER. If you look at the informal specifications of the Database schema in figure 8 below, you would find that the variable "station" is used in several files as the name of a record. Sometimes "station" is a STATION_NAME and sometimes it is a STATION_NUMBER. With informal specifications there is no way to check to see if you are comparing a station

name to a station number. With formal methods each variable will be specified with it's proper datatype so mismatched datatypes cannot be processed together.

4.2.1.2.6. Data For Ever Manual Validation Event Table

Each time a hydrologist validates data, he/she is considered to be performing a "validation event". This event is recorded in the Manual Validation Event Table. Each event usually validates several measurements. If you look at the Measurement Table, you will find the date, time and process of the data that has been validated. You will also find the datatype that was validated. In the Measurement Table, the date will be within the bounds of the beginning and ending date in the Manual Validation Event Table. Any measurement may be validated any number of times. This means that there may be several entries in the Manual Validation Event Table. Only the last validation is recorded in the Measurement Table. The other validation events may be found by searching the Manual Validation Event and matching the station name, datatype and date range.

4.2.1.2.7. Data For Ever Person Table

The Person Table is simply a list of all the hydrologists that are allowed to make changes to the system. It includes the hydrologist's name, login name and password.

4.2.1.2.8/9/10. Data For Ever Validation Process Table

The Validation Process Table is simply a list of the different methods of validating data. The Estimation Method Table is simply a list of the different methods of estimating data

The Reason Value Missing Table is simply a list of the different reason for which a measurement may be missing.

4.2.2 Database schema

There are many methods of representing how data exists in a computer. In section 3.2.2.1.1. and 3.2.2.1.2. above, I have shown (in figures 4 and 5) how the data is represented using a listing that represents the actual data. The development team at ENP has used a format they call "Data For Ever – Database Schema" to represent the data in the Data For Ever database. This is an informal method of depicting a data file. The result of reverse engineering in Z is to write a structural diagram called a **schema**. This is very unfortunate because the word "schema" is used by the development team at ENP to describe the informal diagram that is used to represent the data in a database and it is used to describe the notation used in formal Z specifications. I will make every effort to distinguish the difference between the two. I will refer to the normal schema as a **database schema** and a formal schema as a **Z schema**. Another twist, that makes the dual definition of the word schema difficult, is that the word schema represents two different structures that represent the same thing. In short, we have two schemas that represent the structure of the data, but we can have a normal schema (database schema) or a formal schema (Z schema). A database schema represents the static definition of a database. A Z schema can represent a database, the change in a database, or the activities that change the state of the system.

4.2.2.1 Raw Data

The development team at ENP has not written a database schema for the raw data file, so I have written one. It is displayed in figure 7 below along with the accompanying parse table. The database schema for the raw data shows how the data is arranged in each record of each xx.dat file, each of which represents one monitoring station. The layout of the data in the raw data files is defined in the Data For Ever file named PARSE. A raw data file is a list of records. Each record is a list of numeric fields separated by commas i.e. each field is a number in ASCII format separated by commas with no imbedded blanks. As mentioned in 4.2.1.1. above, these numbers are:

The first number is always the record type.

The second number is the year and is four characters long.

The third number is the Julian day of the year and is1,2, or 3 characters long.

The fourth number is the time in military format except there is no colon (:) and there is no leading zero on the hours before 1000.

The intermediate numbers are the measurements and are defined in the Parse Table in the Data For Ever Database.

The last number, for all of our stations, is the station number, but it is treated as any other measurement even though it is a label.

The record is terminated by a carriage return.

Raw Data, Data Base Schema

xx.dat	
record_type	char 1
comma	char 1
year	char 4
comma	char 1
julian_ date	char 3
comma	char 1
time if time < 1000	char 4
if time > 959	char 3
comma	char 1
measurement(1)	float
comma	char 1
measurement(2)	float
comma	char 1
•••	
comma	char 1
measurement(n)	float
comma	char 1
station	integer
carriage return	char 1

Parse Table

Parse	
station	char 6
record_type	integer
position	integer
datatype	char 25

Figure 7
Database schema of the Raw_data file for a monitoring station,
With accompanying Parse table schema.

4.2.2.2. Data For Ever

The development team at ENP has written a database schema for the Data For Ever database. It is displayed in figure 8 below. The database schema for the Data For Ever database shows how the data is arranged in each record of each of the Data For Ever files, each of which represents one table. The whole Data For Ever database represents all the monitoring stations in the Everglades. The layout of the data in the Data For Ever database files is defined in the several files (tables). Each table is a list of records. Each record has several fields. I will now define each field in detail for the tables that we are interested in (ten tables).

4.2.2.2.1. Data For Ever Measurement Table

The measurement table is the center of all activity and all the other tables exist only to support the measurement table (and the measurement backup table). The "measurement_value" field is the measurement value that came from the raw data. Each other field consists of one of the measurement's labels or descriptors. As mentioned in 4.2.1.2.1. above, the several fields in each records of the Measurement Table are:

The first field is the station name and it can be up to 12 characters long

The second field is the name of the type of data of which the measurement consists.

The third field is the date the measurement was taken.

The fourth field is the time of day the measurement was taken.

The fifth field is the actual measurement itself. This is it. Everything revolves around this measurement. All the measurements as a whole make up the measurement table and the measurements make up the whole study of hydrology.

The sixth field is the estimation method that used to either 1/. Insert a value for the measurement if it is missing, 2/. Change the value if the hydrologist didn't like the original value, or 3/. Blank if the hydrologist thought the value was ok.

The seventh field is the reason for which the value is missing if the value is missing.

The eighth field is the date that the measurement was last validated.

The ninth field is the name of the person that last validated the measurement.

The tenth field is the name of the validation process that was used to validate the measurement.

4.2.2.2.2. Data For Ever Station Table

The station table contains a list of every monitoring station in the Everglades that collects hydrology data. The first and second field are the name and number of the station and the name is the same as the first field in the measurement table. The name and number are unique for each station and virtually never change. The name is also the same as the xx in the name (xx.dat) of the raw data file in the raw data database. As mentioned in 4.2.1.2.2. above the several fields in each records of the Station Table are:

The first field is the station name and it can be up to 12 characters long. There is one entry for each monitoring station and no duplicates.

The second field is the station number and it can be up to 6 characters long. There is one entry for each monitoring station and no duplicates. This is the number that is the last number in each raw data record for all of the stations that we are working on. It is set with DIP switches at the monitoring station and entered as a constant in this table. Hopefully they match but sometimes they don't. The switches can

become corroded or switched by accident. This record and the previous record form the first part of the key between the raw data file "station number" and the Data For Ever "station name". Generally speaking, the Raw Data database uses the numbers and Data For Ever database uses the names.

The third field is the name of the person that is responsible for this station.

There are several other records in this table. Each one describes some attribute of the station. Since we are not going to use any of them, it is not necessary to define them in Z specifications.

4.2.2.2.3. Data For Ever Parse Table

The parse table is the major part of the key for transferring data from the raw data files to the Data For Ever system. As shown in figure 7 above and figure 8 below, the several fields for each record in the Parse Table are:

The first field is the "station number" and it can be up to 6 characters long.

The second field is the "record type" and is a integer number that tells what kind of record we are looking at in the raw data. This number is entered into this file of the Data For Ever database, and hard coded into the firmware in the computer at each monitoring station.

The third field is the "position" of the data in the raw data record.

The fourth field is the "datatype" of the data in the raw data record.

4.2.2.2.4. Data For Ever Station_Datatype Table

The Station Datatype Table contains a list of every datatype for every monitoring station in the Everglades that collects hydrology data. The first field is the name of the station and is the same as the first field in the measurement table. It is also the same as the name (xx.dat) of the file in the raw data database. As mentioned in 4.2.1.2.4. above the several fields in each records are:

The first field is the station name and it can be up to 12 characters long. There is one entry for each datatype for each monitoring station and no duplicates.

The second field is the name of the type of data of which the measurement consists.

The third field is the expected_count, which is the number of measurements that are expected each day.

The fourth field is a "y" or an "n" and tells whether the expected number of counts for each measurement should be counted.

The fifth and sixth field are the upper and lower bounds on the measurement.

The seventh field is a "y" or an "n" and tells whether the bounds for each measurement should be checked.

The eighth field is a "y" or an "n" and tells whether the measurement should be charted in the daily report.

4.2.2.2.5 Data For Ever Datatype Table

The Datatype Table contains a list of every datatype in the Everglades hydrology data. Each record lists a datatype along with the type of units that it represents. The datatype of the datatype is not listed. The several fields in the records are:

The first field is the name of the type of data of which the measurement or object consists.

The second field is the name of the units that the measurement represents. This record is not used if the datatype is not a measurement, i.e. labels do not have units.

The third field is a "y" or an "n" and tells whether the measurement should be graphed.

The fourth field is a "y" or an "n" and tells whether the scale of the graph should start at zero.

4.2.2.2.6. Data For Ever Manual Validation Event Table

The Manual Validation Event Table is a list of each validation session (event) that was undertaken by any hydrologist. The several fields in each records are:

The first field is the name of the if the hydrologist that conducted the validation.

The second field is the date the validation was conducted.

The third field is the time of day the validation was conducted.

The fourth field is the station name.

The fifth field is the datatype of the data that is being validated.

The sixth and seventh field are the beginning and ending dates of the data that is validated in the validation session.

The eighth field is the name of the validation process that was used in the validation process.

4.2.2.2.7. Data For Ever Person Table

The Person Table is a list of all the hydrologists that can use the Data For Ever system. The several fields in each records are:

The first field is the login name.

The second field is the hydrologists' actual name.

The third field is the hydrologists' password.

4.2.2.2.8. Data For Ever Validation Process Table

The Validation Process Table is a list of each kind of validation process that can be used in the Everglades hydrology system. There is only one field and this field contains the name of the validation process. If this thesis is ever implemented it will have to be added to this list. It will probably be called something like "Automatic validation".

4.2.2.2.9. Data For Ever Estimation Method Table

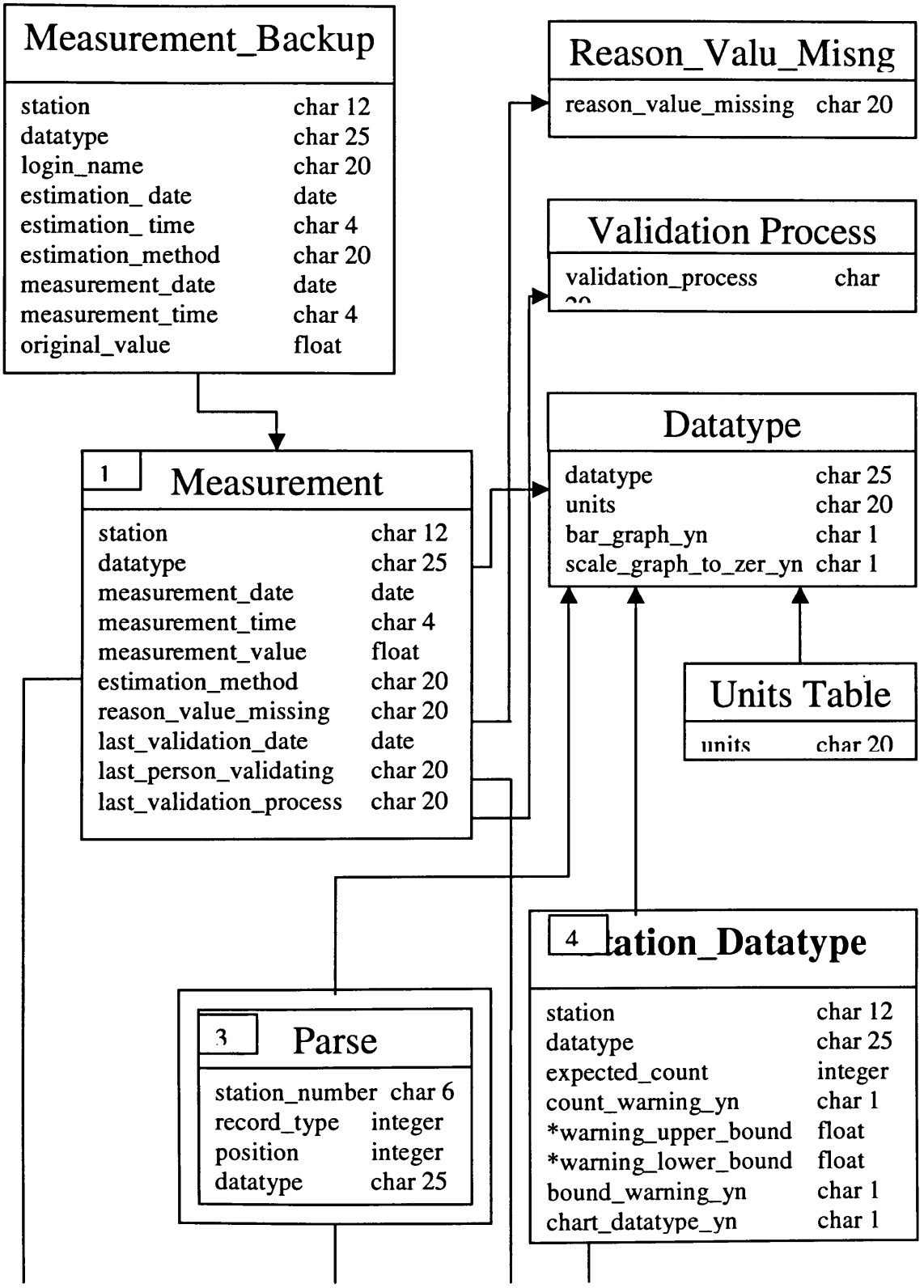
The Estimation Method Table is a list of each kind of estimation process that can be used in the Everglades hydrology system. There is only one field and this field contains the name of the estimation process.

4.2.2.2.10. Data For Ever Reason Value Missing Table

The Reason Value Missing Table is a list of every reason that a measurement value could be missing. There is only one field and this field contains the name of the reasons.

Data For Ever Data Base Schema

* An asterisk in front of the field neans it is a new field that I added.



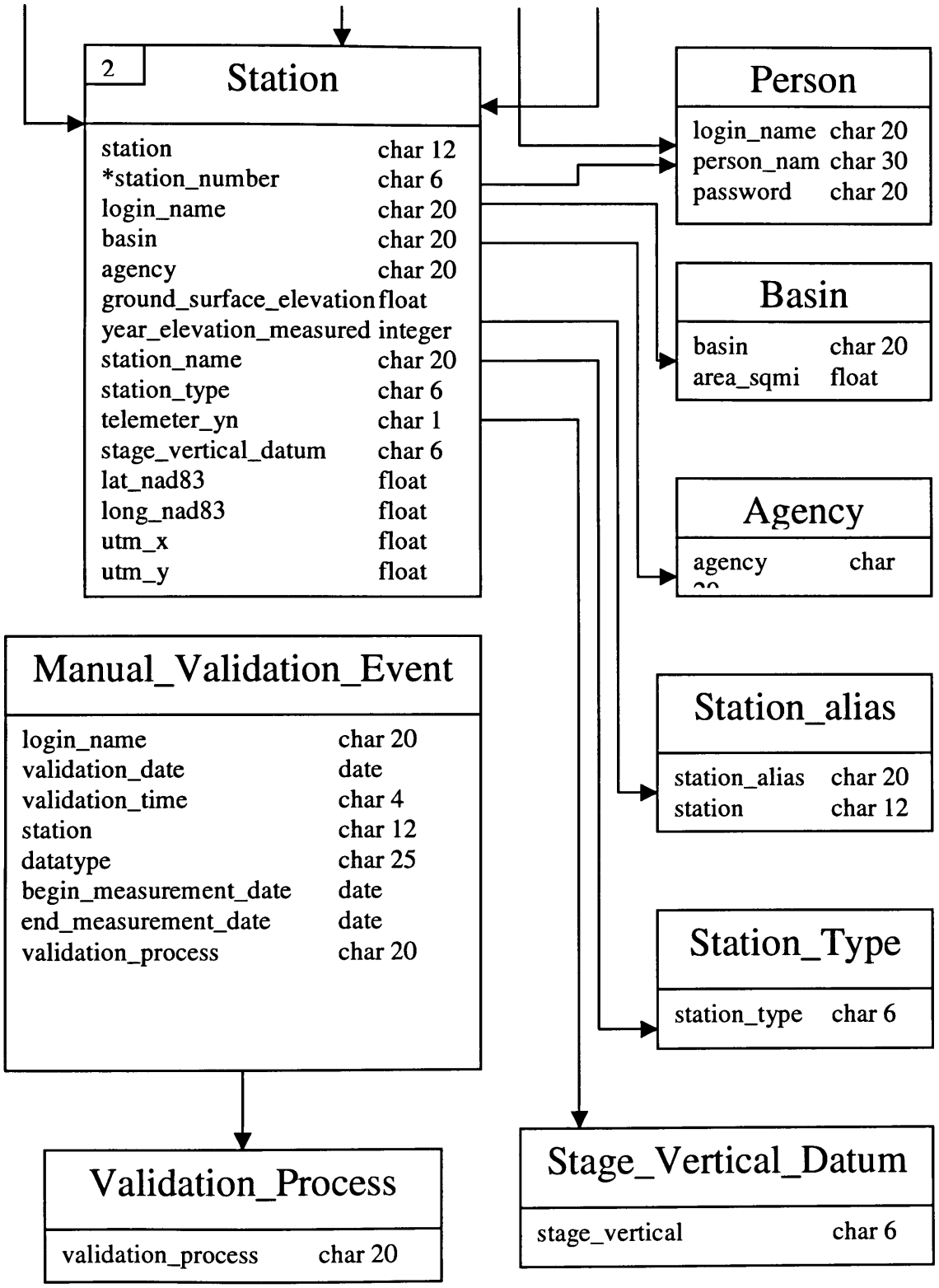


Figure 8
Database schema or block diagram for the Data For Ever system

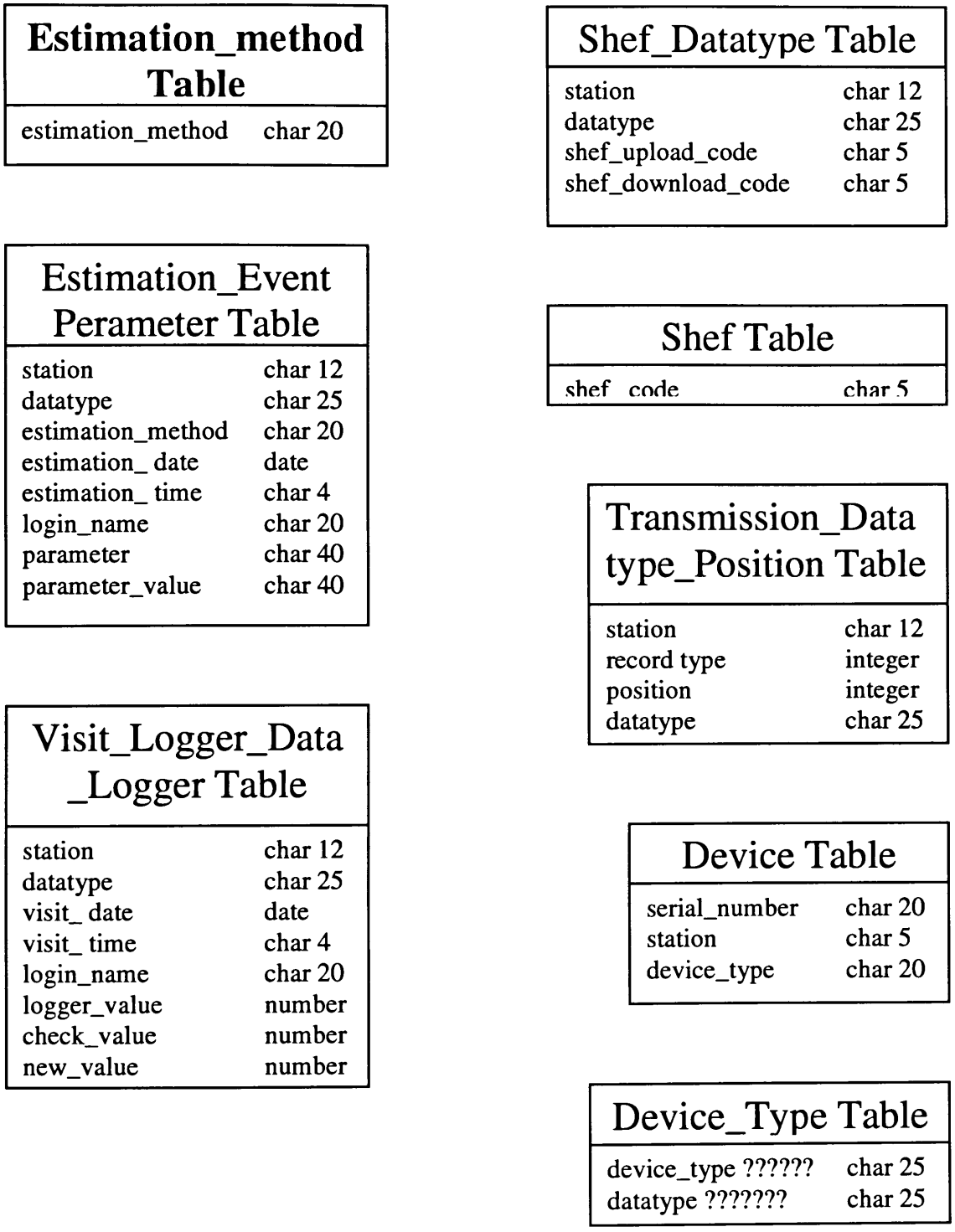


Figure 8a
Database schema or block diagram for the Data For Ever system
We do not use these files so they are not integrated into the schema.

4.2.3. Z specifications of the databases

To specify the data in Z specifications, we must first declare each object (piece of data) much as we would in normal declarations. An object can be a simple single piece of data or several pieces of data concatenated together. We will use an $\times$ as a concatenation operator. As mentioned above in 4.1.1., the basic types of data are CHAR, INTEGER, DATE, and FLOAT. These can be used separately or concatenated to another datatype. The "colon, colon, equal sign" operator is used in BNF to mean "is of type". The format for this is:

TYPE ::= TYPE | TYPE $\times$ TYPE

This type of statement is used outside the Z schema to produce new types of data. This type of equation is recursive, so it may be applied to the result of itself. The convention used in this thesis is that an acronym such as "xx" that begins with a small letter is an object and an acronym such as "XX" is a type of object.

Once we have produced all the different datatypes we will ever use, we will declare the variables in Z format within the Z schema. The format for this is:

variable name : TYPE

We will also need to specify the constraints as invariants. Invariants are logical statements that are always true. When you state an invariant, you are saying that this is the way it has to be done all the time. The format for this is:

boolean statement = TRUE

The "= TRUE" is omitted but it is understood that all invariants are always true. In an invariant the, the "=" means "equal" Later when we are defining the system operations the "=" will mean "replace".

To begin with the whole database is called the "Hydrology System". The Hydrology System can be divided into two major parts, the Raw Data Database and the Data For Ever Database. To link these two database systems together we can use two Z specifications as follows:

Ξ Raw Data Database
Ξ Data For Ever Database

The first statement says that we are linking the Raw Data Database into whatever Z construct we are writing. The Ξ symbol means that we are going to reference the databases but not change them. If we are going to change any of the data in the database we would use the Δ operator instead of the Ξ. These two statements are put together in a form called a Z schema as presented in figure 9.

These are the only constructs that will be used in this thesis. I have tried to keep it simple so any normal programmer can understand it. If you get too fancy, nobody can understand the schema unless they have an in depth understanding of set theory.

4.2.3.1. Z specifications of the Raw Data

This file is the most difficult file to define because each record can be a different length. There are two reasons for this. The first is that there can be a different number of measurements in each record, and the second is that the individual field can be of different lengths.

The objects that compose a raw data record are listed in 4.2.1.1. and 4.2.2.1. above. They are listed here again with the Z specification for each. Each record is composed of several fields terminated by a carriage return. Each field holds a label or a measurement. These fields can be divided into two groups/sets of fields. The first set is composed of four labels which makes up the header. The second set is the measurements and there can be any number of measurements. The only restriction on the number of measurements you can have is dictated by the maximum size of the record. The maximum size of the record is 256 characters. It has always been assumed that this is big enough to handle any record that will ever be constructed by any monitoring stations out in the field in the Everglades. If you think this is a little risky, I agree and I will say more about this in the conclusions.

First of all we can define each set of fields, then each field separately. We will then be able to define a complete raw data record. Finally we can define a raw data file and the complete raw data set.

The different kinds of fields are record type, year, Julian date, measurement time, and raw data measurement value. In each record there is one instance of each of the first four. This makes up the header. The rest of the record is composed of one or more measurements. The definitions of these fields in BNF are:

```
RECORD_TYPE       ::=  INTEGER
YEAR              ::=  INTEGER
JULIAN_DATE       ::=  INTEGER
MEASUREMENT_TIME  ::=  INTEGER
RAW_MEASUREMENT   ::=  FLOAT
```

Because each field is separated from it's surrounding fields by commas, I have defined the comma so there will be no confusion.

COMMA ::= CHAR 1

The two groups can then be defined as:

RAW_DATA_HEADER ::= RECORD_TYPE × COMMA × YEAR × COMMA × JULIAN_DATE × COMMA × MEASUREMENT_TIME × COMMA

ALL_MEASUREMENTS ::= RAW_MEASUREMENT × CR | RAW_MEASUREMENT × COMMA × ALL_MEASUREMENTS

You should notice that the type ALL_MEASUREMENTS can be any length. This is what makes the definition of this record so complicated. The real stickler is that the variable length group of fields is in the middle of the record.

We can now define a complete raw data record as a header and it's measurements. Notice that the terminating carriage return is imbedded in the ALL_MEASUREMENTS group.

RAW_DATA_RECORD ::= RAW_DATA_HEADER × ALL_MEASUREMENTS

The final step is to define a file as a set of records and the complete raw data set as a set of files.

RAW_DATA_FILE ::= RAW_DATA_ RECORD × END_OF_FILE | RAW_DATA_ RECORD × RAW_DATA_FILE

RAW_DATA ::= RAW_DATA_FILE | RAW_DATA_FILE × RAW_DATA_FILE

We will now define a variable for each of the datatypes in Z format. This is so we can define the constraints under which the data must exist. The names of each variable should indicate what it stand for. These Z specifications are as follows:

Ξ Parse Table

raw_data_record	:	RAW_DATA_RECORD
raw_data_header	:	RAW_DATA_HEADER
record_type	:	RECORD_TYPE
year	:	YEAR
julian_date	:	JULIAN_DATE
measurement_time	:	MEASUREMENT_TIME
all_measurements	:	ALL_MEASUREMENTS
raw_measurement	:	RAW_MEASUREMENT
raw_data_file	:	RAW_DATA_FILE
raw_data	:	RAW_DATA

You should notice that the Parse Table has been linked. That is because we need the variable "position" which is defined in the Parse Table. Now we can define the constraints under which the data must be maintained as follows:

record_type	>=	1
1995 <= year	<=	2001
1 <= julian_date	<=	366
MOD(measurement_time, 100)	<=	59
0 <= measurement_time	<=	2400
raw_measurement	=	all_measurements[position]

raw_data = { <Sh1.dat>, <Sh2.dat >, ••• •••, <Zm9.dat >, <Zm10.dat > }

These statements are put together in a form called a Z schema which is presented in figure 10. Figure 10a is the schema that we will use in this thesis. Figure 10b is an alternative schema. It is a little shorter but has no phonetic variables. That makes it more difficult to read.

4.2.3.2. Z specifications of Data For Ever

The objects that compose The Data for Ever database are the several tables that are displayed in figure 8 above. We do not address every table in Data For Ever so I will not define them all in Z specifications. I will only define the tables that we address. The tables that we address are: 1/. the Measurement Table, 2/. the Station Table, 3/. the Parse Table, 4/. The Station Datatype Table, the 5/. Datatype Table, 6/. the Manual Validation

Event Table, 7/. the Person Table, the 8/. Validation Process Table, 9/. the Estimation Method Table, and 10/. the Reason Value Missing Table. Each table is contained in one file, so for this discussion table and file are synonymous. We can define the entire Data For Ever database system in BNF as a set of tables as:

DATA_FOR_EVER ::= MEASUREMENT_TABLE × STATION_TABLE ×
PARSE_TABLE × STATION_DATATYPE_TABLE × DATATYPE_TABLE ×
MANUAL_VALIDATION_EVENT_TABLE × PERSON_TABLE ×
VALIDATION_PROCESS_TABLE × ESTIMATION_METHOD_TABLE ×
REASON_VALUE_MISSING_TABLE × •••

We will now discuss each table in detail. First we will declare what type of data each field is composed of. This will be done with a Z declaration. We can then define the exact values that the data may have. This is done with a Z invariant. The invariant will define the range within which each data module must exist. If the data module is defined in another file it will not have to be defined again. The other file will have to be linked in. These links to these tables are put together in a Z schema which is presented in figure 11.

4.2.3.2.1. Z specifications of the Measurement table for Data For Ever

The main table is the measurement table. The fields that compose a measurement table record are listed in 4.2.2.2. above. They are listed here again with the Z specification for each. There are ten fields in each record. All of the fields are defined in other tables, therefore we must not define any of those fields in this table. We do, however, have to link in each other table that defines one of our fields. We will first define a complete measurement table record in BNF, as follows:

MEASUREMENT_TABLE_RECORD ::= STATION_NAME × DATATYPE ×
MEASUREMENT_DATE × MEASUREMENT_TIME ×
MEASUREMENT_VALUE × ESTIMATION METHOD ×

REASON_VALUE_MISSING × LAST_VALIDATION_DATE ×
LAST_PERSON_VALIDATING × LAST_VALIDATION_PROCESS
MEASUREMENT_DATE ::= CHAR 10
MEASUREMENT_VALUE ::= FLOAT

We will now define each of these fields separately or link to the table in which they are defined.

All of the fields are defined in other tables, so we merely have to link in the table in which they are defined. The measurement time field comes in two parts so it has to be combined, and the measurement value has to be indexed by the position field of the Parse Table.

The first field is the name of the station, and is defined in the Station Table, so we need only to link to the Station Table. This is done with the following Z specification:

Ξ Station Table && a single station name

The second field is the name of the type of data of which the measurement consists. It is defined in the Station Datatype Table, so we need only to link in the Station Datatype Table. This is done with the following Z specification:

Ξ Station Datatype Table && a single datatype

The third field is the date the measurement was taken. The format is a little different than it was in the raw data, but we will handle that with an invariant after we have completed the declarations. It is defined in one of the Raw Data files in the Raw data Database, so we need only to link to the Raw Data Database. This is done with the following Z specification:

Ξ Raw Data Database && the date, time & measurement

We will, however have to declare a variable to hold the date. The standard format that is used in the ENP hydrology system is "mm-dd-yyyy" which is 10 alpha characters. This is represented in Z specification as:

measurement_date : MEASUREMENT_DATE

We will also assume that there is a subroutine named CONVERT_RAW_DATE( ) which will convert the raw data year and Julian date to the Data For Ever date. This subroutine will be used in the invariant.

The fourth field is the time of day the measurement was taken. This is in the same format as in the raw data so no modifications need to be made. The raw data file has already been linked in so there is no need to do it again.

The fifth field is the actual measurement itself. Everything revolves around this measurement. All the measurements as a whole make up the measurement table and the measurements make up the whole study of hydrology. This is in the same format as in the raw data so no modifications need to be made. The raw data file has already been linked in so there is no need to do it again. We do, however need to link in the Parse Table because the Parse Table tells us where each measurement is in the raw data record. This is done with the following Z specification:

Ξ Parse Table && the position of the measurement

The sixth field is the estimation method that was used to either 1/. Insert a value if the measurement was missing, 2/. Change the value if the hydrologist didn't like the original value, or 3/. Blank if the hydrologist thought the value was ok. This value is defined in

the Estimation Method Table, so we must link in the Estimation Method Table with the following Z specification.

Ξ Estimation Method Table && a list of all the estimation methods

The seventh field is the reason for which the value is missing if the value is missing. This value is defined in the Reason Value Missing Table, so we must link in the Reason Value Missing Table with the following Z specification.

Ξ Reason Value Missing Table && a list of all the reasons why a value could be missing

The eighth, ninth, and tenth field have to do with the manual validation process. Although we are doing validation, we are not doing manual validation. We are doing automatic validation. Our automatic validation is not necessarily a real validation because it is not done by a hydrologist, it is done by a machine. In the future we may make an entry in this file, but for now, we will not. We will, however link in the Manual Validation Event Table for completeness.

Ξ Manual Validation Event Table

Finally, the complete measurement table is just a collection of measurement table records.

MEASUREMENT_TABLE ::= MEASUREMENT_TABLE_RECORD × EOF |
MEASUREMENT_TABLE_RECORD × MEASUREMENT_TABLE

We must now specify a variable that can hold a measurement table record. We need to do this so we can specify the constraints by which the data must follow. We do

this by stating the constraints in invariant clauses. The measurement table record variable is:

measurement_table_record : MEASUREMENT_TABLE_RECORD

The measurement table cannot hold anything you want. It can only hold certain information. These constraints on what can be put in each field can be defined in a set of invariants. The constraints are:

For the third field, the measurement date, the year and Julian date are retrieved from the raw data. They are then converted into a date that is used in the Data For Ever system. We have linked in the Raw Data Database, so now we can use any of the variables that are defined there. The Z specification to do this is:

measurement_date = CONVERT_RAW_DATE(year, julian_date)

For the fifth field, the measurement value is retrieved from the raw data file via the parse table. The Raw Data Database is already linked and so is the Parse Table. We do not need to link them in again so we can define the measurement value in Z specifications as:

measurement_value = raw_data_record [position + 4]

The last validation date and process are retrieved from the Manual Validation Event Table. This can be specified in Z as:

last_validation_date = validation_date
last_validation_process = validation_process

These specifications are put together in a Z schema which is presented in figure 12.

4.2.3.2.2. Z specifications of the Station table for Data For Ever

The Everglades hydrology system is composed in part of hydrology monitoring stations. Each station is described in detail in the station table. Each record in the Station Table represents one station. The objects that compose a station record in the Station Table are listed in 4.2.2.2.2. above. These objects/fields are listed here again with the Z specification for each. The structure of each field in a Station Table record is defined in BNF as:

STATION_TABLE_RECORD ::= STATION_NAME × STATION_NUMBER ×
LOGIN_NAME × 2CHAR 20 × FLOAT × INTEGER × CHAR 20 ×
CHAR 6 × CHAR 1 × CHAR 6 × 4 FLOAT
STATION_NAME ::= CHAR 12

The first field is the station name and it can be up to 12 characters long. It is declared as:

station_name : STATION_NAME

and defined as:

station_name = {<"Sh1">, <"Sh2">, ••• •••, <"Zm9">, <"Zm10"> }

The definition above is a small part of the actual definition. The actual definition includes about 600 station names. Each of these names is unique.

The second field is the station number and it can be up to 6 characters long. It is declared as:

station_number : CHAR 6

and defined as:

station_number >= 0

Each station has a unique number that is assigned to it. This gives two ways to identify a station. In general, the number is used in the raw data and the alphanumeric acronym is used by Data For Ever.

The third field is the name of the person that is responsible for this station. This field is defined in the Person Table along with a list of all the possible names. Each name is the name of a hydrologist who is eligible to work on the hydrology data. To be able to use this name we only have to link in the Person Table as follows:

Ξ Person Table && a single person that can log in

There are several more records in the station table, however, we will not refer to them so I will not define them.

•••

•••

We will then be able to define the entire station table in BNF as:

STATION_TABLE ::= STATION_TABLE_RECORD × EOF |
STATION_TABLE_RECORD × STATION_ TABLE

These specifications are put together in a Z schema which is presented in figure 13.

4.2.3.2.3. Z specifications of the Parse table for Data For Ever

The Parse Table holds the key to the process of moving the raw data into Data For Ever. The objects that compose a parse record are listed in 4.2.2.2.3. above. They are listed here again with the Z specification for each. We can now define a Parse Table record in BNF by declaring a datatype for each field as follows:

PARSE_TABLE_RECORD ::= STATION_NUMBER × RECORD_TYPE × POSITION × DATATYPE

The first field is the station number and it can be up to 6 characters long. It is defined in the Station Table, so we need only to link in the Station Table. This is done with the following Z specification:

Ξ Station Table && a single station name

The second field is the type of the record that we are looking at in the raw data. It has to be declared as an integer in the BNF definition of the parse table record. We can then declare the Z specifications as an integer as follows:

RECORD_TYPE ::= INTEGER
record_type : RECORD_TYPE && a single record type

and defined as:

record_type >= 1

The third field is the position of the data in the raw data file. It has to be declared as an integer in the BNF definition of the parse table record, then it can be declared in Z specifications as an integer as follows:

POSITION ::= INTEGER
position : POSITION

and defined as:

position >= 1

The fourth record is the datatype and it can be up to 25 characters long. It is defined in the Station Datatype Table, so we need only to link in the Station Datatype Table as follows:

Ξ Station Datatype Table && a single datatype

The only remaining object in the parse table is the complete Parse Table which is a concatenation of several Parse Table records and defined as:

PARSE_TABLE ::= PARSE_TABLE_RECORD × EOF |
PARSE_TABLE_RECORD × PARSE_TABLE

These specifications are put together in a Z schema which is presented in figure 14.

4.2.3.2.4. Z specifications of the Station Datatype table for Data For Ever

The objects that compose a record in the Station Datatype Table are listed in 4.2.2.2.4. above. They are listed here again with the Z specification for each. We can now define a Station Datatype Table record in BNF by declaring a datatype for each field as follows:

STATION_DATATYPE_TABLE_RECORD ::= STATION_NAME ×
DATATYPE × EXPECTED_COUNT × COUNT_WARNING_YN ×
WARNING_UPPER_BOUND × WARNING_LOWER_BOUND ×
BOUND_WARNING_YN × CHART_DATATYPE_YN

The first field is the station name and it can be up to 12 characters long. There is one entry for each datatype for each monitoring station and no duplicates. It is defined in the Station Table, so we need only to link in the Station Table. This is done with the following Z specification:

Ξ Station Table && a single station name

The second field is the name of the type of data of which the measurement consists. Be careful!! The datatype of the data is datatype. There is one entry for each datatype for each monitoring station and no duplicates. It is defined in the Datatype Table, so we need only to link in the Datatype Table. This is done with the following Z specification:

Ξ Datatype Table && a single datatype

The third record is the expected_count, which is the number of measurements that are expected each day (usually 24). We will declare the variable type in BNF and then a variable that will hold one expected count with the following Z specification:

EXPECTED_COUNT ::= INTEGER
expected_count : EXPECTED_COUNT

and define all the possible expected counts as:

expected_count = {<3>, <6>, <12>, <24>}

The fourth field is a "y" or an "n" and tells whether the expected number of counts for each measurement should be counted. We will declare the variable type in BNF and then a variable that will hold one yes/no indicator with a Z specification:

COUNT_WARNING_YN ::= CHAR 1
count_warning_yn : COUNT_WARNING_YN

and define it as:

count_warning_yn = {<y>, <n>}

The fifth and sixth field are the upper and lower bounds for each measurement. We will declare the variable type in BNF and then declare a variable that will hold one of each with Z specification:

WARNING_UPPER_BOUND ::= FLOAT
WARNING_LOWER_BOUND ::= FLOAT
warning_upper_bound : WARNING_UPPER_BOUND
warning_lower_bound : WARNING_LOWER_BOUND

We will not define any values for these variables because they can be about any value and they will be defined later by one of the new modules.

The seventh and eighth fields are similar to the fourth field. That is they are a "y" or an "n" and tell whether the bounds for each measurement should be checked and

charted. We will declare the variable type in BNF and then declare a variable that will hold one of each with Z specifications:

BOUND_WARNING_YN	::=	CHAR 1
CHART_DATATYPE_YN	::=	CHAR 1
bound_warning_yn	:	CHAR 1
chart_datatype_yn	:	CHAR 1

and define them as:

bound_warning_yn	=	{<y>, <n>}
chart_datatype_yn	=	{<y>, <n>}

We will now be able to define an entire station datatype table record in BNF as:

STATION_DATATYPE_TABLE ::= STATION_DATATYPE_TABLE_RECORD × EOF | STATION_DATATYPE_TABLE_RECORD × STATION_DATATYPE_TABLE

These specifications are put together in a Z schema which is presented in figure 15.

4.2.3.2.5. Z specifications of the Datatype table for Data For Ever

The objects that compose a record in the Datatype Table are listed in 4.2.2.2.5. above. They are listed here again with the Z specification for each. We can now define a Datatype Table record in BNF by declaring a datatype for each field as follows:

DATATYPE_TABLE_RECORD ::= DATATYPE × UNITS × BAR_GRAPH_YN × S_G_2_0_YN

The first field is the name of the type of data of which a piece of data consists. Be careful!! The datatype of the data is datatype. There is one entry for each datatype for any monitoring station and no duplicates. If two stations have the same datatype they both use the same entry in this table. It can be 25 characters long and we can declare it in BNF and Z specifications as:

DATATYPE ::= CHAR 25
datatype : DATATYPE

and define it as an invariant in Z specifications as:

datatype = {"Battery voltage" | ••• ••• | "Salinity"}

The second field is the name of the units that the measurement represents and it can be up to 20 characters long. There is one entry for each datatype and no duplicates. If several monitoring station use the same datatype, it will have the same units. The BNF and Z specification are:

UNITS ::= CHAR 20
units : UNITS

and the definition is:

units = {<Volts>, ••• •••, {<Miliseamans>}

The third field is a "y" or an "n" and tells whether the piece of data should be graphed. The BNF and Z specification are:

BAR_GRAPH_YN ::= CHAR 1
bar_graph_yn : BAR_GRAPH_YN

and the definition is:

bar_graph_yn = {<y>, <n>}

The fourth field is a "y" or an "n" and tells if the graph (if one is made) should start at zero or at the lowest value. The BNF and Z specification are:

S_G_2_0 ::= CHAR 1
scale_graph_to_zero_yn : BAR_GRAPH_YN

and the definition is:

scale_graph_to_zero_yn = {<y>, <n>}

We will now be able to define an entire datatype table record in BNF as:

DATATYPE_TABLE ::= DATATYPE_TABLE_RECORD × EOF |
DATATYPE_TABLE_RECORD × DATATYPE_TABLE

These specifications are put together in a Z schema which is presented in figure 16.

4.2.3.2.6. Z specifications of the Manual Validation Event table for Data For Ever

At present, there is no automatic validation. Everything has to be validated by hand. There are several validation processes that can be used and each time any data is validated, an entry is made in the Manual Validation Event Table. The Manual Validation Event Table has eight fields as mentioned in 4.2.2.2.6. above. I will repeat them here along with the Z specifications to declare and define them. The BNF declaration for a Manual Validation Event Table Record is:

MANUAL_VALIDATION_EVENT_TABLE_RECORD ::= LOGIN_NAME ×
VALIDATION_DATE × VALIDATION_TIME × STATION_NAME ×
DATATYPE × BEGIN_MEASUREMENT_DATE ×
END_MEASUREMENT_DATE × VALIDATION_PROCESS

The first field is the name of the hydrologist that conducted the validation. This field is defined in the Person Table along with a list of all the possible names. Each name is the name of a hydrologist who is eligible to work on the hydrology data. To be able to use this name we only have to link in the Person Table as follows:

Ξ Person Table && a single person that can log in

The second field is the date the validation was conducted and is represented in BNF and Z as:

VALIDATION_DATE ::= CHAR 10
validation_date : VALIDATION_DATE

It is defined by the system date at the time of validation and is defined as an invariant in Z as:

validation_date =: DATE( )

The third field is the time of day the validation was conducted and is represented in BNF and Z as:

VALIDATION_TIME ::= CHAR 4
validation_time : VALIDATION_TIME

It is defined by the system time at the time of validation and is defined as an invariant in Z as:

validation_time =: TIME( )

The fourth field is the station name. The station names are defined in the Station Table, so we need to link in the Station Table then we can define the station name in this database.

Ξ Station Table && a single station name

The fifth field is the datatype of the data that is being validated. The datatypes are defined in the Datatype Table so we need to link in the Datatype Table.

Ξ Datatype Table && a single datatype

The sixth and seventh fields are the beginning and ending dates of the data that is validated in the validation session. They are declared locally and are defined in the new modules being added to the system. The BNF and Z specifications for this are:

BEGIN_MEASUREMENT_DATE ::= CHAR 10
END_MEASUREMENT_DATE ::= CHAR 10
begin_measurement_date : BEGIN_MEASUREMENT_DATE
end_measurement_date : END_MEASUREMENT_DATE

The restrictions on these dates are that they cannot be before January first 1995 or after yesterdays date. Also, the ending date cannot be before the beginning date. The Z specifications for these constraints are:

$$01\text{-}01\text{-}1995 \leq begin\_measurement\_date \leq end\_measurement\_date \leq (DATE() - 1)$$

The eighth field is the name of the validation process that was used to validate the measurements. The list of validation processes is listed in the Validation Process Table. We need to link in the Validation Process Table as follows:

Ξ Validation Process Table

We can now declare a complete Manual Validation Event Table in BNF as follows:

MANUAL_VALIDATION_EVENT_TABLE ::=
MANUAL_VALIDATION_EVENT_TABLE_RECORD × EOF |
MANUAL_VALIDATION_EVENT_TABLE_RECORD ×
MANUAL_VALIDATION_EVENT_TABLE

These specifications are put together in a Z schema which is presented in figure 17.

4.2.3.2.7. Z specifications of the Person table for Data For Ever

The Person Table is simply a list of all the people that are eligible to sign on the ENP Hydrology system and alter the data. These names are accompanied by their login name and password. The Person Table has three fields as mentioned in 4.2.2.2.7. above and no foreign references. I will repeat them here along with the Z specifications to declare and define them. The BNF declaration for a Person Table Record is:

PERSON_TABLE_RECORD ::= LOGIN_NAME × PERSON_NAME ×
PASSWORD

The first field is the login name and can have up to 20 characters. The BNF and Z declaration for this field are:

LOGIN_NAME ::= CHAR 20
login_name : LOGIN_NAME && a single person's login name

and the definition are:

login_name = {<Anderson>, <Koten>, •••}

The second field is the person's actual name and can be up to 30 characters long. The BNF and Z declaration for this field is:

PERSON_NAME ::= CHAR 30
person_name : PERSON_NAME && a single person that can log in

and the definition is:

person _name = {<Gordon Anderson>, <Kevin Koten>, •••}

The third field is the password and can have up to 20 characters. The BNF and Z declaration for this field are:

PASSWORD ::= CHAR 20
password : PASSWORD

and the definition is:

password = {<Gordo>, <Happy>, •••}

We can now declare a complete Person Table in BNF as follows:

PERSON_TABLE ::= PERSON_TABLE_RECORD × EOF |
PERSON_TABLE_RECORD × PERSON_TABLE

These specifications are put together in a Z schema which is presented in figure 17.

4.2.3.2.8. Z specifications of the Validation Process table for Data For Ever

The Validation Process Table is, very simply, one list of all the available validation processes that are available in the ENP Hydrology system. The table has one field, which is the validation_process field, and it can contain up to 20 characters. The BNF and Z declaration for this field are:

REASON_VALUE_MISSING	::=	CHAR 20
validation_process	:	REASON_VALUE_MISSING

and the definition is:

validation_process = {<Eyeball>, <Average>, •••}

We do not have to define a record for this file because there is only one field in the whole record and we can access that variable using the file name.

4.2.3.2.9. Z specifications of the Estimation Method table for Data For Ever

The Estimation Method Table is similar to the Validation Process Table. It is a simple list of all the estimation methods that are available in the ENP Hydrology system. The table has one field, which is the estimation_method field, and it can contain up to 20 characters. The BNF and Z declaration for this field are:

ESTIMATION_METHOD	::=	CHAR 20
estimation_method	:	ESTIMATION_METHOD

and the definition is:

estimation_method = {<Linear interpolation>, <Insert null measurements>, <Nearest neighbor>, <Linear interpolation drift>,•••}

We do not have to define a record for this file because there is only one record in the whole file and we can access that variable using the file name.

4.2.3.2.10. Z specifications of the Reason Value Missing table for Data For Ever

The Reason Value Missing Table is similar to the Validation Process Table, and the Estimation Method Table. It is a simple list of all the reasons why a measurement value could be missing. The table has one field, which is the reason_value_missing field, and it can contain up to 20 characters. The BNF and Z declaration for this field are:

REASON_VALUE_MISSING ::= CHAR 20
reason_value_missing : CHAR 20

and the definition is:

reason_value_missing = {<Accumulated Later>, <Device Malfunction>, <Power Failure>, •••}

We do not have to define a record for this file because there is only one record in the whole file and we can access that variable using the file name.

4.2.3.3. Z schemas of the databases

We can now put all the declarations together to form a Z schema that represents the two databases. Figure 9 shows the Raw Data database and the Data For Ever database defined in Z specifications.

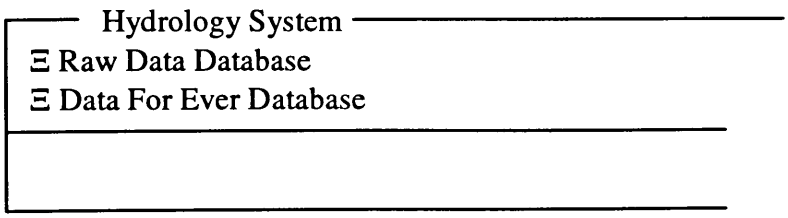


Figure 9
Z schema for the entire Hydrology system

```
RAW_DATA_RECORD      ::=  RAW_DATA_HEADER ×
                                  ALL_MEASUREMENTS
 RAW_DATA_HEADER     ::=  RECORD_TYPE × COMMA × YEAR × COMMA
                          × JULIAN_DATE × COMMA ×
                          MEASUREMENT_TIME × COMMA
   RECORD_TYPE       ::=  INTEGER
   YEAR              ::=  INTEGER
   JULIAN_DATE       ::=  INTEGER
   MEASUREMENT_TIME  ::=  INTEGER
   COMMA             ::=  CHAR 1
 ALL_MEASUREMENTS    ::=  RAW_MEASUREMENT × CR |
                          RAW_MEASUREMENT × COMMA ×
                          ALL_MEASUREMENTS
   RAW_MEASUREMENT   ::=  FLOAT
RAW_DATA_FILE        ::=  RAW_DATA_ RECORD × END_OF_FILE |
                          RAW_DATA_ RECORD × RAW_DATA_FILE
RAW_DATA             ::=  RAW_DATA_FILE | RAW_DATA_FILE ×
                          RAW_DATA_FILE
```

Raw Data Database

```
Ξ Parse Table
raw_data_record      :  RAW_DATA_RECORD
  raw_data_header    :  RAW_DATA_HEADER
    record_type      :  RECORD_TYPE
    year             :  YEAR
    julian_date      :  JULIAN_DATE
    measurement_time :  MEASUREMENT_TIME
  all_measurements   :  ALL_MEASUREMENTS
    raw_measurement  :  RAW_MEASUREMENT
raw_data_file        :  RAW_DATA_FILE
raw_data             :  RAW_DATA
```

```
                record_type          >=    1
1995    <=      year                 <=    2001
1       <=      julian_date          <=    366
MOD(measurement_time, 100)           <=    59
0       <=      measurement_time     <=    2400
raw_measurement  =  all_measurements[position]
raw_data    = { <Sh1.dat>, <Sh2.dat >, ••• •••, <Zm9.dat >, <Zm10.dat > }
```

Figure 10a
Z schema for the Raw Data databases

```
RAW_DATA_RECORD      ::=  RAW_DATA_HEADER × ALL_MEASUREMENTS
  RAW_DATA_HEADER    ::=  4 (INTEGER × COMMA)
    COMMA            ::=  CHAR 1
  ALL_MEASUREMENTS   ::=  FLOAT × CR | FLOAT × COMMA × ALL_MEASUREMENTS
RAW_DATA_FILE        ::=  RAW_DATA_ RECORD × END_OF_FILE |
                          RAW_DATA_ RECORD × RAW_DATA_FILE
RAW_DATA             ::=  RAW_DATA_FILE | RAW_DATA_FILE × RAW_DATA_FILE
```

Raw Data Database

```
Ξ Parse Table
raw_data_record      :  RAW_DATA_RECORD
  raw_data_header    :  RAW_DATA_HEADER
  all_measurements   :  ALL_MEASUREMENTS
    raw_measurement  :  FLOAT
raw_data_file        :  RAW_DATA_FILE
raw_data             :  RAW_DATA
```

```
                raw_data_header[1]  >=   0
1995  <=        raw_data_header[2]  <=   YEAR(DATE( ))
1     <=        raw_data_header[3]  <=   366
MOD(raw_data_header[3], 100)      <=   59
0     <=        raw_data_header[4]  <=   2400
raw_measurement  =  all_measurements[position]
raw_data  = { <Sh1.dat>, <Sh2.dat >, ••• •••, <Zm9.dat >, <Zm10.dat > }
```

Figure 10b
Alternate Z schema for the Raw Data databases

Note:
This is an alternate schema of the raw data database. You should notice that the fields in the header do not have a two level declaration. This makes it easier to define the database but harder to access the individual fields. For example, you have to write "raw_data_header[1]" instead of "record_type".

****
****

DATA_FOR_EVER ::= MEASUREMENT_TABLE × STATION_TABLE × PARSE_TABLE × STATION_DATATYPE_TABLE × DATATYPE TABLE × MANUAL_VALIDATION_EVENT_TABLE × PERSON _TABLE × VALIDATION_PROCESS _TABLE × ESTIMATION_METHOD _TABLE × REASON_VALUE_MISSING _TABLE

Data For Ever Database

Ξ Measurement Table
Ξ Station Table
Ξ Parse Table
Ξ Station Datatype Table
Ξ Datatype Table
Ξ Manual Validation Event Table
Ξ Person Table
Ξ Validation Process Table
Ξ Estimation Method Table
Ξ Reason Value Missing Table
data_for_ever : DATA_FOR_EVER

Figure 11
Z schema for the Data For Ever databases

****
****

****
****

MEASUREMENT_TABLE_RECORD ::= STATION_NAME × DATATYPE ×
MEASUREMENT_DATE × MEASUREMENT_TIME ×
MEASUREMENT_VALUE × ESTIMATION METHOD ×
REASON_VALUE_MISSING × LAST_VALIDATION_DATE ×
LAST_PERSON_VALIDATING × VALIDATION_PROCESS
MEASUREMENT_DATE ::= CHAR 10
MEASUREMENT_VALUE ::= FLOAT
MEASUREMENT_TABLE ::= MEASUREMENT_TABLE_RECORD × EOF |
MEASUREMENT_TABLE_RECORD × MEASUREMENT_TABLE

Measurement Table

Ξ Station Table	&& a list of the station names
Ξ Station Datatype Table	&& a list of the datatypes
Ξ Raw Data Database	&& the dates, times, & measurements
Ξ Parse Table	&& the position of the measurement
Ξ Estimation Method Table	&& a list of the estimation methods
Ξ Reason Value Missing Table	&& a list of the reasons a value could be missing
Ξ Manual Validation Event Table	&& a list of the validation events
measurement_date :	MEASUREMENT_DATE
measurement_value :	MEASUREMENT_VALUE
last_validation_process :	VALIDATION_PROCESS
measurement_table_record :	MEASUREMENT_TABLE_RECORD

measurement_date	=	CONVERT_RAW_DATE(year, julian_date)
last_validation_date	=	validation_date
last_validation_process	=	validation_process
measurement_value	=	raw_data_record[position + 4]

Figure 12
Z schema for the Measurement Table of the Data For Ever databases
See Figure 5 for an example

****

****

STATION_TABLE_RECORD ::= STATION_NAME × STATION_NUMBER × LOGIN_NAME × 2CHAR 20 × FLOAT × INTEGER × CHAR 20 × CHAR 6 × CHAR 1 × CHAR 6 × 4 FLOAT

STATION_NAME ::= CHAR 12

STATION_NUMBER ::= CHAR 6

STATION_ TABLE ::= STATION_TABLE_RECORD × EOF |
STATION_TABLE_RECORD × STATION_ TABLE

Station Table

Ξ Person Table			&& a single person that can log in
station_name	:	STATION_NAME	&& a single station name
station_number	:	STATION_NUMBER	&& a single station number
basin	:	~~~	
agency	:	~~	
•••			
•••			
•••			
utmx			
utmy			
station_table_record	:	STATION_TABLE_RECORD	

station_name = {<"Sh1">, <"Sh2">, ••• •••, <"Zm9">, <"Zm10"> }
station_number >= 0

Figure 13
Z schema for the Station Table of the Data For Ever databases

Example of Station Table

Station	Station_number	Login_name	[4]	[5]	[6]						[15]
"Sh1"	"68"	"Anderson"									
"Sh2"	"69"	"Koten"									
"Sh3"	"70"	"Anderson"									
"Sh4"	"71"	"Koten"									
"Sh5"	"72"	"Anderson"									
Sta?	station_number										

****

PARSE_TABLE_RECORD ::= STATION_NUMBER × RECORD_TYPE × POSITION × DATATYPE

RECORD_TYPE ::= INTEGER

POSITION ::= INTEGER

PARSE_TABLE ::= PARSE_TABLE_RECORD × EOF |
PARSE_TABLE_RECORD × PARSE_TABLE

Parse Table

Ξ Station Table			&& a single station name
Ξ Station Datatype Table			&& a single datatype
record_type	:	RECORD_TYPE	&& a single record type
position	:	POSITION	&& a single position
parse_table_record	:	PARSE_TABLE_RECORD	
parse_table	:	PARSE_TABLE	

record_type	>=	1
position	>=	1

Figure 14
Z schema for the Parse Table of the Data For Ever databases

****

Example of Parse Table

Station_number	Record_type	Position	Datatype
"68"	"1"	"1"	
"68"	"1"	"2"	
station_number	"1"	"3"	"Battery_voltage"
station_number	"1"	"4"	"Station_number"
station_number	"2"	"1"	
station_number	"2"	"2"	
station_number	"2"	"3"	"Battery_voltage"
station_number	"2"	"4"	"Station_number"
station_number	"6"	"1"	"Rain"
station_number	"6"	"2"	"Station_number"
station_number	"7"	"1"	
station_number	"7"	"2"	
station_number	"7"	"3"	
station_number	"7"	"4"	
station_number	"7"	"5"	
station_number	"7"	"6"	
station_number	"7"	"7"	
station_number	"7"	"8"	
station_number	"7"	"9"	"Battery_voltage"
"68"	"7"	"10"	"Station_number"
"69"			
station_number			

****

****

STATION_DATATYPE_TABLE_RECORD ::= STATION_NAME ×
DATATYPE × EXPECTED_COUNT × COUNT_WARNING_YN ×
WARNING_UPPER_BOUND × WARNING_LOWER_BOUND ×
BOUND_WARNING_YN × CHART_DATATYPE_YN
EXPECTED_COUNT ::= INTEGER
COUNT_WARNING_YN ::= CHAR 1
WARNING_UPPER_BOUND ::= FLOAT
WARNING_LOWER_BOUND ::= FLOAT
BOUND_WARNING_YN ::= CHAR 1
CHART_DATATYPE_YN ::= CHAR 1
STATION_DATATYPE_TABLE ::= STATION_DATATYPE_TABLE_RECORD
× EOF | STATION_DATATYPE_TABLE_RECORD ×
STATION_DATATYPE_TABLE

Station Datatype Table

Ξ Station Table		&& a single station name
Ξ Datatype Table		&& a single datatype
expected_count	:	EXPECTED_COUNT
count_warning_yn	:	COUNT_WARNING_YN
warning_upper_bound	:	WARNING_UPPER_BOUND
warning_lower_bound	:	WARNING_LOWER_BOUND
bound_warning_yn	:	BOUND_WARNING_YN
chart_datatype_yn	:	CHART_DATATYPE_YN
station_datatype_table_record	:	STATION_DATATYPE_TABLE_RECORD
station_datatype_table	:	STATION_DATATYPE_TABLE

expected_count	=	{<3>, <6>, <12>, <24>}
count_warning_yn	=	{<y>, <n>}
bound_warning_yn	=	{<y>, <n>}
chart_datatype_yn	=	{<y>, <n>}

Figure 15
Z schema for the Station Datatype Table of the Data For Ever databases

****

****

DATATYPE_TABLE_RECORD ::= DATATYPE × UNITS × BAR_GRAPH_YN × S_G_2_0

DATATYPE ::= CHAR 25

UNITS ::= CHAR 20

BAR_GRAPH_YN ::= CHAR 1

S_G_2_0 ::= CHAR 1

DATATYPE_TABLE ::= DATATYPE_TABLE_RECORD × EOF | DATATYPE_TABLE_RECORD × DATATYPE_TABLE

Datatype Table

datatype	:	DATATYPE && a single datatype
units	:	UNITS
bar_graph_yn	:	BAR_GRAPH_YN
scale_graph_to_zero_yn	:	S_G_2_0
datatype_table_record	:	DATATYPE_TABLE_RECORD
datatype_table	:	DATATYPE_TABLE

datatype	=	{<"Battery voltage">, <"Salinity">, ••• }
units	=	{<"Voltage">, <"Miliseamons">, ••• }
bar_graph_yn	=	{<"y">, <"n">}
scale_graph_to_zero_yn	=	{<"y">, <"n">}

Figure 16
Z schema for the Datatype Table of the Data For Ever databases

****

****

MANUAL_VALIDATION_EVENT_TABLE_RECORD ::= LOGIN_NAME ×
VALIDATION_DATE × VALIDATION_TIME × STATION_NAME ×
DATATYPE × BEGIN_MEASUREMENT_DATE ×
END_MEASUREMENT_DATE × VALIDATION_PROCESS

VALIDATION_DATE ::= CHAR 10
VALIDATION_TIME ::= CHAR 4
BEGIN_MEASUREMENT_DATE ::= CHAR 10
END_MEASUREMENT_DATE ::= CHAR 10

MANUAL_VALIDATION_EVENT_TABLE ::=
MANUAL_VALIDATION_EVENT_TABLE_RECORD × EOF |
MANUAL_VALIDATION_EVENT_TABLE_RECORD ×
MANUAL_VALIDATION_EVENT_TABLE

Manual Validation Event Table

Ξ Person Table && a single person that can log in
Ξ Station Table && a single station name
Ξ Datatype Table && a single datatype
Ξ Validation Process Table
validation_date : VALIDATION_DATE
validation_time : VALIDATION_TIME
begin_measurement_date : BEGIN_MEASUREMENT_DATE
end_measurement_date : END_MEASUREMENT_DATE
manual_validation_event_table_record :
MANUAL_VALIDATION_EVENT_TABLE_RECORD
manual_validation_event_table : MANUAL_VALIDATION_EVENT_TABLE

---

validation_date =: DATE( )
validation_time =: TIME( )
01-01-1995 <= begin_measurement_date <= end_measurement_date <=
(DATE( ) – 1)

Figure 17
Z schema for the Manual Validation Even Table of the Data For Ever databases

****

****

PERSON_TABLE_RECORD ::= LOGIN_NAME × PERSON_NAME × PASSWORD
LOGIN_NAME ::= CHAR 20
PERSON_NAME ::= CHAR 30
PASSWORD ::= CHAR 20
PERSON_TABLE ::= PERSON_TABLE_RECORD × EOF | PERSON_TABLE_RECORD × PERSON_TABLE

Person Table

login_name	:	LOGIN_NAME && a single persons login name
person_name	:	PERSON_NAME && a single person that can log in
password	:	PASSWORD && a single persons password
person_table_record	:	PERSON_TABLE_RECORD && a single persons login name, name and password
person_table	:	PERSON_TABLE && all the people

login_name = {<Anderson>, <Koten>, •••}
person _name = {<Gordon Anderson>, <Kevin Koten>, •••}
password = {<Gordo>, <Happy>, •••}

Figure 18
Z schema for the Person Table of the Data For Ever databases

****

VALIDATION_PROCESS_TABLE ::= VALIDATION_PROCESS × EOF | VALIDATION_PROCESS × VALIDATION_PROCESS_TABLE
VALIDATION_PROCESS ::= CHAR 20

Validation Process Table

validation_process	:	VALIDATION_PROCESS && a single validation process
validation_process_table	:	VALIDATION_PROCESS_TABLE && all processes

validation_process = {<Eyeball>, <Average>, •••}

Figure 19
Z schema for the Validation Process Table of the Data For Ever database

****

****

ESTIMATION_METHOD_TABLE ::= ESTIMATION_METHOD × EOF |
ESTIMATION_METHOD × ESTIMATION_METHOD_TABLE
ESTIMATION_METHOD ::= CHAR 20

Estimation Method Table

estimation_method	:	CHAR 20 && a single estimation method
estimation_method_table	:	ESTIMATION_METHOD_TABLE && all

estimation_method = {<Linear interpolation>, <Insert null measurements>, <Nearest neighbor>, <Linear interpolation drift>, •••}

Figure 20
Z schema for the Estimation Method Table of the Data For Ever database

****

REASON_VALUE_MISSING_TABLE ::= REASON_VALUE_MISSING × EOF |
REASON_VALUE_MISSING × REASON_VALUE_MISSING_TABLE
REASON_VALUE_MISSING ::= CHAR 20

Reason Value Missing Table

reason_value_missing	:	REASON_VALUE_MISSING && a single
reason_value_missing_table	:	REASON_VALUE_MISSING_TABLE && all

reason_value_missing = {<Accumulated Later>, <Device Malfunction>, <Power Failure>, •••}

Figure 21
Z schema for the Reason Value Missing Table of the Data For Ever databases

****
****

4.3. Forward Engineering (Software Architecture) for additions to the existing Data For Ever software

We are now ready to proceed with adding new elements to the system. The databases have been defined in formal specifications so we can refer to the elements of these databases as operands in formal logical statements. The fact that the database was defined using reverse engineering procedures is irrelevant. If this had been a new project, the final outcome would have been the same (except that hopefully it would have been a better design). We will now summarize the several processes that are required to perform validation. We will then define each module. We will first give a rough English definition, then we will give a informal (normal) version, and finally we will define each module in formal specifications. The English version will be a very casual version that anyone should be able to understand. The informal version will be the version that is frequently used by the military. It is composed of several "shall" clauses that define what each module will do. In the final step we will define each process in Z specifications.

4.3.1. The Validation process

The additions (enhancements) to the ENP hydrology system will be implemented by specifying the high level design for the verification modules in Z specifications. Each function that will be performed will be defined in an informal manner using standard definition methods. No new databases are planned, at this time, as this is a running system with no particular problems. There are, however, a couple of fields (variables)

that will have to be added to the "Station_Datatype" table and one field that will have to be added to the "Station" table. The two fields to be added to the Station_Datatype table are "warning_upper_bound" and "warning_lower_bound". The field to be added to the Station table is "station_number". Once these fields have been added, the functions to be performed will be 1/. Enter the station number, 2/. Enter the upper and lower bound, 3/. Validate the labels in the Raw Data, and 4/. Validate the measurements in the "Measurement" table of the Data For Ever database.

4.3.2. Validation Modules definitions

As mentioned in 4.3.1. above, there are 4 activities that have to be performed. Each of these activities (except the first) will be implemented as a separate function in a separate module or program. I will list the requirements for each, then I will list them again with the Z specifications and finally I will put the Z specifications together in a Z schema.

4.3.2.1. Enter the station number Validation Modules definition

There will be no module to execute the first function, "Enter the station number". This is because it is a simple function and it is a one-time process. I expect an operator to enter these numbers by hand. There is also a module available that can change any station attributes, so this process can be executed with existing software or at worst case a very small modification to the existing software.

4.3.2.2. Enter the upper and lower bound Validation Modules definition

The second function, "Enter the upper and lower bound", will be an ongoing activity. Even after the values are entered, they will have to be changed from time to time. We will define a process that will enter these values. This process will perform the following functions:

a. Enter the upper bound for any measurement at any station.

b. Enter the lower bound for any measurement at any station.

4.3.2.3. Validate the labels, Validation Modules definitions

The third function, "Validate the labels", will run automatically daily plus it will be able to be run at any time on any set or subset of data in the raw data set. We will define a process that will run in the morning shortly after the data is downloaded from each station. This process will perform the following functions:

a. Check to see if the station file (xx.dat) exists, i.e. to verify that the station did download last night.

b. Check the station ID to see that it is a valid station ID.

c. Check the line type to see that it is a valid line type.

d. Check the station ID to see that it is the same for the entire download from each station as it is being read from the field.

e. Check the Date to see that it is yesterday's date.

f. Check the Time to see that it is consistent. That is, the time starts at 12:00 midnight and ends at 11:59 PM and there are no missing or extra time intervals

The first three parameters that are checked (line type, station signature, date and time stamp) are labels. These labels are exact parameters and should never have any flexibility to their form or content. They do, however, have to be checked to ensure that the hardware, firmware and software are working properly. These parameters must be checked while the data is still in the raw data mode. Once the data has been parsed into Data For Ever, there is nothing to check these parameters against. This is a very touchy matter because if the data is parsed into Data For Ever with any of these labels in error, there is no way to catch the error in the future. It should also be noted very clearly that if the data is parsed into Data For Ever with errors in any of the labels, there may be other data that is already there and is the correct data. If the correct data and the incorrect data have matching labels, there is no way to tell which is the correct data and which is the incorrect data. The only way to tell is to go back to the raw data with the correct label and correct the error, or assume that you can change the dataset that matches the current raw data. The first solution is very complicated and can lead to confusion and compounded errors. The second solution is correcting errors assuming the errors are currently not the other data (a circular proof). You would be far better off not to parse any data into Data For Ever until you have checked out the labels as much as humanly or computationally possible.

4.3.2.4. Validate the measurements, Validation Modules definitions

The final function, "Validate the measurements" will also run automatically daily, plus it will be able to be run at any time on any set or subset of data in the Data For Ever System. We will define a process that will run in the morning shortly after the data is parsed from the raw data to the Data For Ever database. These next sets of parameters, to be verified, are measurements. These measurements measure the condition of the water in the swamp. We will define and design a process that will measure the following measurements.

a. Check the battery voltage to see that it is within range.

b. Check the rain.

c. Check the water level.

d. Check the salinity.

e. Check the temperature.

They are to be checked to see if they are within the limits that are defined as warning_upper_bound and warning_lower_bound as recorded in the Station_Datatype Table. They can be checked at any time but it makes more sense to check them once they have reached Data For Ever. Once they have reached Data For Ever the original value will be saved in the "Measurement_Backup" table, so that if a new value is estimated you can go back to the original and start over again. We will then be able to check and/or change any measurement at any time. The checking process is called ***validation*** and the changing process is called ***estimation***. We are only doing validation in this thesis.

4.3.3. Z specifications

We will now list each of the functions with the requirements for each and the Z specifications that define that process.

4.3.3.1. Entering the "station_number" field in the "Station" table

Again this is a one shot deal and once it has been performed it will never change. There is no need to write a program to perform this action so none will be defined. We will expect that these values will be entered by hand or entered using the existing software.

4.3.3.2 Enter the "warning_upper_bound" field and the "warning_lower_bound" field in the "Station_Datatype" table.

The name of this module will be "Enter Limits". It will perform the following functions. When execution is initiated:

a. Enter the upper bound for any measurement at any station.

b. Enter the lower bound for any measurement at any station.

c. Replace the original values of the upper and lower bounds with the new.

The measurements are tracked by datatype. Each different type of measurement has a different datatype. All the different possible data types are listed in the Datatype Table. The Station Datatype Table is a listing of each datatype for each station. Each different datatype at each different station has it's own upper and lower bound. That means that if there are about 600 stations and each station could measure about 10 different measurements, there could be about 6000 entries in the Station Datatype Table.

We can first specify the execution of the module in military format, which is an English version that specifies each action of the program as a ***shall*** clause. This definition of this module would then be:

When execution is initiated: The program ***shall*** prompt the operator to enter the station for which he/she wishes to check or change the measurement limits. The operator ***shall*** then pick a station, out of the Station Table. The program ***shall*** then prompt the operator for the datatype which he/she wishes to check or change. The operator ***shall*** then pick a datatype out of the Station Datatype Table. The program ***shall*** then display the upper and lower bounds for that measurement (datatype) at that station. The program ***shall*** then prompt the operator to enter a new value for either the upper or lower bounds. The program ***shall*** then replace the old upper and/or lower bounds with the new ones if new value(s) were entered.

We will now define the program in Z specifications. We must first specify the inputs. The station and datatype in Z specifications are:

1st.	input_sta?	: STATION_NAME	
2nd.	input_type?	: DATATYPE	
3rd.	Ξ Station Table		&& a list of station names
4th.	Δ Station Datatype Table		&& a list of station names and datatypes

The first statement says that "input_sta?" is an input (because of the ?) variable and it can represents a station name that is the type that is listed in the Station Table. It can represent a station name because we have defined it with the same datatype as that of the station name in the Station Table. We must, therefore, link to the Station Table. The second statement says that "input_type?" is an input (because of the ?) variable and it

represents a datatype because it is the same type that is used for the datatype in the Station Datatype Table. We must, therefore, link to the Station Table.

Notes:

1/. The Station Table is linked in with the no change operator and the Station Datatype Table is linked in with the change operator. At this point they both only need the no change operator, but later we are going to need to change the values of some of the data in the Station Datatype Table.

2/. The datatype is originally defined in the Datatype Table. By referring to the Station Datatype Table, we do not get every datatype, we only get the datatypes that are present at the station that specified.

This satisfies the underlined portion of the requirements.

a. Enter the upper bound for any measurement at any station.

b. Enter the lower bound for any measurement at any station.

We will now define a local variable to represent a Station Table Record and a Station Datatype Table Record.

5th. a : STATION_TABLE_RECORD
6th. b : STATION_DATATYPE_TABLE_RECORD

The fifth statement says that "a" is a variable that can hold any Station Table Record. This statement is unnecessary because a global variable "station_table_record" has already been defined. The reason that I have added this new local variable is because it is shorter and easier to handle and this definition gives a local view of the data definition. Also, it cannot be changed by any outside influence. The sixth statement is similar to the third for the "station_datatype_table_record".

We now need to define two local variables to represent the upper and lower bounds that are to be entered by the operator and two more local variables to represent the "warning_upper_bound" and "warning_lower_bound" that are currently stored in the Station Datatype Table.

7^{th}.	input_hi?	:	WARNING_UPPER_BOUND
8^{th}.	input_lo?	:	WARNING_LOWER_BOUND
9^{th}.	hi	:	WARNING_UPPER_BOUND
10^{th}.	lo	:	WARNING_LOWER_BOUND

Note:

1/. The datatypes "WARNING_UPPER_BOUND" and "WARNING_LOWER_ BOUND" are defined in the Station Datatype Table.

Again, the last two local variables are unnecessary, but local variables make a program more manageable, readable and secure. This satisfies the underlined portion of the requirements.

a. Enter the upper bound for any measurement at any station.

b. Enter the lower bound for any measurement at any station.

We have already stated that we will input any station and any datatype. What we really meant is that we will input any station that exists in the system and after we have picked an existing station, we will enter any one of the datatypes that exist for that station. To make sure that these two restrictions are met, we will include the following two statements. These statements are invariants and are the following:

11^{th}. $\exists a: a \in station\_table \cap a[1] = input\_sta?$
12^{th}. $\exists b: b \in station\_datatype\_table \cap b[1] = input\_sta? \cap b[2] = input\_type?$

The eleventh statement says that variable "input_sta" must always be in the first (because of the [1]) field of a record of the Station Table (because of the "$\exists a: a \in$ "). This will

probably be implemented with a dropdown menu so the operator will have to pick a station that actually exists. There are other ways to implement this function but, because of this statement, the programmer will be forced to write the program so the operator can only choose a station that is a valid station. The twelfth statement insures that only a valid datatype can be chosen.

We can now perform the replacement of the upper and lower limits. This will be done with two steps. The first step will be to fetch the upper and lower bounds from the database. This process is not necessary but is a good idea because these original values can be presented when the operator is requested for new values. The second step is to replace the upper and lower bounds with the new values. The statements to implement this process are:

13^{th}. ($\exists$b: b $\in$ station_datatype_table $\cap$ b [1] = input_sta? $\cap$ b [2] = input_type?) $\Rightarrow$
hi = b[5] $\cap$ lo = b[6]

14^{th}. b[5]' = input_hi?

15^{th}. b[6]' = input_lo?

16^{th}. (b[1]' = b[1] $\wedge$ b[2]' = b[2] $\wedge$ b[3]' = b[3] $\wedge$ b[4]' = b[4] $\wedge$ b[5]' = input_hi? $\wedge$
b[6]' = input_lo? $\wedge$ b[7]' = b[7] $\wedge$ b[8]' = b[8]) $\wedge$

17^{th}. Station Datatype Table' = Station Datatype Table – b $\cap$ b'

The thirteenth statement picks the upper and lower bounds out of the database and puts them in the local variables. The fourteenth and fifteenth statements dictate that the new values are stored back in the database. This satisfies the requirement:

c. Replace the original values of the upper and lower bounds with the new.

I have added an extra segment to demonstrate how an error condition would be reported if the operator were allowed to input an improper station name or datatype. These statements should never be executed after the code has been debugged but may be a

helpful debugging tool. They also serve as an example of how an error should be expressed in Z format. The statements are:

18^{th}. msg! : MSG
19^{th}. $\neg$($\exists$b: b $\in$ station_datatype_table $\cap$ b [1] = input_sta? $\cap$ b [2] = input_type?)
$\Rightarrow$ msg! = "There is no datatype " + type? + " for " + sta? + "."

The eighteenth statement sets up the variable "msg!" as an output (because of the !) and the nineteenth statement actually outputs an error message if an error occurs.

4.3.3.3. Validating the labels in the Raw Data

The name of this module will be "Validate Labels". It will perform the following functions.

a. Input the station.

b. Check to see if the station file (xx.dat) exists, i.e. to verify that the station did download last night.

c. Check the station ID to see that it is a valid station ID.

d. Check the line type to see that it is a valid line type.

e. Check the station ID to see that it is the same for the entire download from each station as it is being read from the field.

f. Check the Date to see that it is yesterday's date.

g. Check the Time to see that it is consistent. That is, the time starts at 12:00 midnight and ends at 11:59 PM and there are no missing or extra time intervals

The labels are first generated at each station out in the field in the Everglades. They are then transmitted to the base station every day early in the morning. They are then parsed into the Data For Ever database where they reside for infamy. If these labels are not

checked before they get to the Data For Ever Database, there is no way ever to check them.

We can first specify the execution of the module in military format, which is an English version that specifies each action of the program as a ***shall*** clause. This program can execute in either automatic mode or work under operator control.

When automatic execution is initiated, the definition of this module would be: The program ***shall*** automatically pick every station, one after the other. For each station it ***shall*** then pick yesterday's date for the date.

When operator control execution is initiated, the program ***shall*** prompt the operator to enter the station for which he/she wishes to check the labels. The operator ***shall*** then pick a station, out of the Station Table. The program ***shall*** then prompt the operator for the date(s) which he/she wishes to check. The operator ***shall*** then enter a date or a range of dates to be checked.

Finally, whether the program is started automatically or under operator control, the program ***shall*** then check all the labels (line type, station signature, date and stamp) , to see if they are correct. If there are any errors in the labels, the program ***shall*** report these errors.

We will now define the program in Z specifications. We must first specify that we will enter the station name. In Z specifications it is:

1st. input_sta? : STATION_NAME
2nd. sta_no : STATION_NUMBER
3rd. Ξ Station Table && a list of station names

Each station has a unique name. Actually each station has two unique names. One name is an alphanumeric cryptic abbreviated name that has some meaning. An

example is "Sh1" which stands for the first station on the Shark River. This station also has a numeric name which is "68". This number is coded into DIP switches at the monitoring station out in the Everglades. In general, the numeric name is used in the raw data files, and the alphanumeric name is used in Data For Ever. The Station Table in Data For Ever is a list of each station and it's attributes. The first two fields of the Station Table are the alphanumeric name and the numeric name respectively. We must, therefore, link to the Station Table. The first statement of our specification says that "input_sta?" is an input (because of the ?) variable and it can represents a station name that is the same type that is listed in the first field of the Station Table. The second statement of our specification says that "sta_no" is a variable and it can represents a station number that is the same type that is listed in the second field of the Station Table. The third statement links in the Station Table so we can reference the variables there. This satisfies the requirement:

a. Input the station.

We have suggested that the station that is input must be a valid station. We will now add a couple of statements to the specifications that insures this to be the case. This statements are the follows:

4th. a : STATION_TABLE_RECORD
5th. ∃a: (a ∈ station_table ∩ a[1] = input_sta?) ∪ "All" = input_sta?

The fourth statement says that "a" is a variable and it can represent a record in the Station Table. The fifth statement says that there exists (because of the ∃) a record (because of the "a") in the station table in the first field (because of the [1]) that contains the station

name that is stored in the variable input_sta?. This statement also allows the operator to enter "All", indicating that all the stations should be checked.

We will now set up a local variable to hold the filename in the of the raw data file for the station we are interested in checking the labels. The statements to do this are:

```
6th.  raw_file   :   CHAR 16
7th.  raw_file = input_sta? +".dat"
```

The sixth statement sets up a variable that holds the complete raw data file name. The station name can be up to 12 characters long. If we add ".dat" to the station name it will require 16 characters, thus the declaration of a variable with 16 characters in it. The seventh statement puts the raw data file name in the local variable.

We can now address the raw data file so we will set up a local variable to hold the contents of a record in the raw data file. We will also have to link in the raw data database to do this. The statements are:

```
8th.  raw_rec    :   RAW_DATA_RECORD
9th.  Ξ Raw Data Database                    && Raw Data Database
```

We can now hold the contents of one raw data record in the local variable "raw_rec".

We will also need to address the Parse Table. To do this we have to link to the Parse Table and define a record that will hold a record of the Parse Table. The statements to do this are:

```
10th. Ξ Parse Table                          && a list of datatypes for each station and
                                                their position in the raw data file
11th. c          :   PARSE_TABLE_RECORD
```

The tenth statement links the Parse Table in to this module. The eleventh statement sets up a local variable that will hold one record of the Parse Table. We are going to use the Parse Table to find the position of the station number in the raw data file. The station

number in the raw data file is treated as a measurement even though we all know that it doesn't measure any aspect of the swamp (i.e. the station number is actually a label). To check the station number, we have to find it just the same way we would find any other measurement. We have to look in the Parse Table for the entry that matches the station number datatype ("Station number"), and raw data record type. There should be one and only one of these. In this record in the parse table we will find the position number that represents the position in the raw data record where we will find the station number. There are easier ways to do this if the database had been set up differently. I will discuss this later in the conclusions. We are stuck with this setup because the system is up and running. Perhaps at a later date it will be advantageous to modify the database, but this is not the time.

We now have to set up two local variables, one to hold the raw data record type and the other to hold the position in the raw data file where we will find the station number. These statements are:

12th. type : RECORD_TYPE
13th. sta_no_pos : POSITION

The twelfth statement declares a variable that can hold the raw data record type (remember that the first field in a raw data record is the record type. Raw data records have record types. Data For Ever records do not). The RECORD_TYPE is declared in the raw data database, and the raw data database is already linked in. The thirteenth statement declares the position in the raw data record that we will find a particular measurement (in this case the station number). The POSITION is declared in the Parse Table, and the Parse Table is already linked in.

One problem is that we cannot check everything. We have to assume that something is sacred and we assume that this is the record type in the raw data file. I will say more about this in the conclusions but for now we assume the record type is correct and then check all the other labels.

The first thing we will check is to see if the raw data file actually exists. Sometimes the radio or computer stops working and no data is downloaded for a station. In this case the file would not exist. The statement to check for the existence of the file is:

14^{th}. (∃raw_file: raw_file ∉ raw_data) ⇒ (msg! = "File " + input_sta? +".dat does not exist")

This fourteenth statement says that if the file does not exist, output (because of the !) a message.

If the file does exist, we can check the labels in every record of the raw data file that we have chosen. The first label that we check is the station number. This is the most difficult one because it is in a different location for each raw data record type. The statement to check this is :

15^{th}. (∃raw_file: raw_file ∈ raw_data) ⇒
((∀raw_rec: raw_rec ∈ raw_file ∩ type = raw_rec[1]) ∩
(∃a: a ∈ station_table ∩ a[1] = input_sta? ⇒ sta_no = a [2]) ∩
(∃c: c ∈ parse_table) ∩ c[1] = sta_no
∩ c[2] = type
∩ b[4] = "Station_number" ⇒ sta_no_pos = b [3]) ∩
(raw_rec[sta_no_pos] ≠ sta_no) ⇒
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field " + raw_rec[station_no_position] + ".")

This statement says:

(∃raw_file: raw_file ∈ raw_data) ⇒

If the file does exist then:

((∀raw_rec: raw_rec ∈ raw_file ∩ type = raw_rec[1]) ∩

For all records in the raw data file and with the record type in the first field:

(∃a: a ∈ station_table ∩ a[1] = input_sta? ⇒ sta_no = a [2]) ∩

if there exists a record in the station table where the first field is equal to the station that the operator input (and there does because that's where we got the station name to start with) then we can pick the station number out of the second field and:

(∃c: c ∈ parse_table) ∩ c[1] = sta_no
∩ c[2] = type
∩ b[4] = "Station_number" ⇒ sta_no_pos = b [3]) ∩

if there exists a record in the Parse Table where the first field is the station number that we just picked out of the station table, the second field is the record type of the raw data record that we are currently checking, and the fourth field has "Station_number" in it, then we can pick the raw data station number position out of field three and:

(raw_rec[sta_no_pos] ≠ sta_no) ⇒
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +"
in field " + raw_rec[station_no_position] + ".")

if we look in the raw data record, in the station number position that we just picked out of the Parse Table and compare the station number in that position in the raw data record, it should match the station number that we picked out of the station table. If these two station numbers do not match we will report an error.

We now only have to check the date. The date in the raw data is divided into two fields. Field two is the year and field three is the Julian date. I will assume that there are three subroutines available. The first is DATE which should be a system function that will give us the current date. The second is YEAR which should give us the year from

any date. The third is JULIAN which should convert the Julian date to a regular date. The next two statements check the date.

16th. (∀raw_rec: raw_rec[2] ≠ YEAR(DATE()-1))) ⇒
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field 2 which is the year field."

17th. (∀c: raw_rec[3] ≠ JULIAN(DATE()-1))) ⇒
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field 3 which is the Julian date field."

The sixteenth statement checks the year and the seventeenth statement checks the date.

4.3.3.4. Validating Data For Ever

The name of this module will be "Validate Measurements". It will perform the following functions.

a. Input the station and datatype.

b. Input the date or range of dates.

c. Check to see if the specified measurements for the specified station are within the range defined in the Station Datatype Table.

This program can execute in either automatic mode or work under operator control. When automatic execution is initiated:

The program ***shall*** automatically pick every station, one after the other. For each station it ***shall*** then pick yesterday's date for the date. It ***shall*** then check every measurement in the raw data to see if they are correct. If the measurements are not within the range defined by the upper bound and the lower bound that is stored in the station datatype table, the program ***shall*** report an error.

When operator control execution is initiated:

It ***shall*** prompt the operator to enter the station for which he/she wishes to check

the measurements. The operator ***shall*** then pick a station, out of the Station Table. The program ***shall*** then prompt the operator for the datatype which he/she wishes to check. The operator ***shall*** then pick a datatype out of the Station Datatype Table. The program ***shall*** then prompt the operator for the date(s) which he/she wishes to check. The operator ***shall*** then enter a date or a range of dates to be checked. The program ***shall*** then check to see if the measurements in the Measurement Table are correct. If the measurements are not within the range defined by the upper bound and the lower bound that is stored in the Station Datatype Table, the program ***shall*** report an error.

We will now define the program in Z specifications. We must first specify the station and datatype in Z specifications as:

1st. input_sta? : STATION_NAME
2nd. Ξ Station Table && a list of station names
3rd. a : STATION_TABLE_RECORD
4th. ∃a: (a ∈ station_table ∩ a[1] = input_sta?) ∪ "All" = input_sta?
5th. input_type? : DATATYPE
6th. Ξ Station Datatype Table && a list of datatypes for each station and their position in the raw data file.
7th. b : STATION_DATATYPE_TABLE_RECORD
8th. ∃b: b ∈ station_datatype_table ∩ b[1] = input_sta? ∩ b[2] = input_type?

The first statement says that "input_sta" is an input (because of the ?) variable and it represents a station name, because we made it the same type as that of the station name field which is listed in the Station Table. The second statement links in the Station Table where the STATION_NAME is declared. The third statement says that "a" is a variable that will hold one station table record. In other words, "a" could represent any record in the Station Table. The Station Table is already linked in so it does not have to be linked in again. The fourth statement says that there exists (because of the ∃) a record in the

Station Table (because of the a) that is a member of (because of the $\in$) the station table, and (because of the $\cap$) the first (because of the [1]) field of that record is the same as the station name that we input. The fifth statement says that "input_type?" is an input (because of the ?) variable and it represents a datatype that is the type that is listed in the Station Datatype Table. Be careful reading this because it is true that you do have a datatype of datatype. These datatypes are defined in the Station Datatype Table which must be linked in as shown in the sixth statement. The seventh statement says that "b" is a variable that will hold one station datatype table record. The station datatype table record is declared in the Station Datatype Table so we do not have to link it in again. The eighth statement says that there exists (because of the $\exists$) a record (because of the b) that is a member of (because of the $\in$) the station datatype table, and (because of the $\cap$) the first (because of the [1]) field of that record is the same as the station that we input and the second (because of the [1]) field of that record is the same as the datatype name that we input. This satisfies the following requirement.

a. Input the station and datatype.

We must now enter the dates.

9th. begin_date? : MEASUREMENT_DATE
10th. end_date? : MEASUREMENT_DATE
11th. da : MEASUREMENT_DATE
12th. time : MEASUREMENT_TIME
13th. Ξ Measurement Table

The ninth statement says that we are to enter the beginning date and it will be stored in the variable "begin_date?". The tenth statement says that we are to enter the ending date and it will be stored in the variable "end_date?". The eleventh statement says that we can hold a measurement date in the variable "da". This will be the day that we

are currently checking. The twelfth statement says that we can hold a time in the variable "time". This will be the time of day that we are currently checking. These datatypes are defined in the Measurement Table which is linked in with the thirteenth statement. This satisfies the requirement.

b. Input the date or range of dates.

We must now find the upper and lower bounds in the database.

14^{th}. hi : WARNING_UPPER_BOUND
15^{th}. lo : WARNING_LOWER_BOUND
16^{th}. $\exists b: b \in$ station_datatype_table $\cap$ b[1] = input_sta? $\cap$ b[2] = input_type? $\Rightarrow$
hi = b[5] $\cap$ lo = b[6]

The fourteenth statement says that "hi" is a variable and it can represent an upper bound because it is the same type that is listed for the "warning_upper_bound" field in the Station Datatype Table. The fifteenth statement says that "lo" is a variable and it can represent a lower bound because it is the same type that is listed for the "warning_lower_bound" field in the Station Datatype Table. The Station Datatype Table is already linked in so we do not have to do it again. The sixteenth statement says that if there is a record "b" that is a member of the station datatype table and the first element/field of that record is the station that we input and the second element/field of that record is the datatype that we input(and it is because we already stated that as an invariant in statement 8), then "hi" is the fifth entry of that field and "lo" is the sixth entry of that record. This satisfies the underlined portion of the requirements.

c. Check to see if the specified measurements for the specified station are within the range defined in the station datatype table.

17^{th}. measurement : MEASUREMENT_VALUE
18^{th}. c : MEASUREMENT_TABLE_RECORD

The seventeenth statement says that "measurement" is a variable and it can represent a measurement value because it is the same type as the fifth element/field (the "measurement" field) in the Measurement Table. The eighteenth statement says that "c" is a variable that can represent any measurement table record.

Now we can finally check the measurement to see if it is within the range that is defined in the Station Datatype Table.

19th. ((∀c: c ∈ measurement_table ∧ begin _date? <= c[3] <= end_date? ∧ c[1] = input_sta? ∧ c[2] = input_type? ∧ da = c[3] ∧ hr = c[4]
) ⇒
((∃b: b ∈ station_datatype_table ∧ b[1] = input_sta? ∧ b[2] = input_type?
hi = b[5] ∧ lo = b[6]
) ⇒
((c[5] >= hi ∨ c[5] <= lo) ⇒
msg! = "There is an error in station " + input_sta? + " for datatype " + input_type? + " on " + da + " at " + time + "in which the measurement is out of range."
)))

The sixteenth statement says that:

((∀c: c ∈ measurement_table
for every measurement table record such that:

begin _date? <= c[3] <= end_date?
the date is within the dates that were input, and:

c[1] = input_sta? ∧ c[2] = input_type?
the station name and the datatype are the same as the input, and:

da = c[3] ∧ hr = c[4]
) the third and fourth field are a date and time, then:

((∃b: b ∈ station_datatype_table
there exists a station table record such that:

b[1] = input_sta? ∧ b[2] = input_type?
the station name and the datatype are the same as the input, and:

hi = b[5] ∧ lo = b[6]

) the fifth and sixth fields are floating point numbers like the upper and lower bounds, then:

((c[5] >= hi ∨ c[5] <= lo) ⇒
if the actual measurement is higher than the upper bound or lower than the lower bound, then: error.

msg! = "There is an error in station " + input_sta? + " for datatype " + input_type? + " on " + da + " at " + time + "in which the measurement is out of range."

)))

This satisfies the requirement.

b. Check to see if the specified measurements for the specified station are within the range defined in the station datatype table.

4.3.4. Z schema

We now have all the modules specified in Z format. We want this put in the form of a Z schema. The following is a Z schema for each module (except the first).

4.3.4.1. Entering the "station_number" field in the "Station" table

This is a one shot deal so there are no Z specifications for it.

4.3.4.2. Entering the limits

Enter Limits

Ξ Station Table		&& a list of station names
Δ Station Datatype Table		&& a list of station names and datatypes
input_sta?	:	STATION_NAME
input_type?	:	DATATYPE
hi	:	WARNING_UPPER_BOUND
input_hi?	:	WARNING_UPPER_BOUND
lo	:	WARNING_LOWER_BOUND
input_lo?	:	WARNING_LOWER_BOUND
msg!	:	MSG
a	:	STATION_TABLE_RECORD
b	:	STATION_DATATYPE_TABLE_RECORD
b'	:	STATION_DATATYPE_TABLE_RECORD

Invariants

$\exists a: a \in$ station_table $\wedge$ a[1] = input_sta?
($\exists b: b \in$ station_datatype_table $\wedge$ b[1] = input_sta? $\wedge$ b[2] = input_type?)

Preconditions

($\exists b: b \in$ station_datatype_table $\wedge$ b [1] = input_sta? $\wedge$ b [2] = input_type?) $\Rightarrow$
hi = b [5] $\cap$ lo = b [6]
($\neg$($\exists b: b \in$ station_datatype_table $\wedge$ b [1] = input_sta? $\wedge$ b [2] = input_type?)
$\Rightarrow$ msg! = "There is no datatype " + type? + " for " + sta? + ".") $\wedge$

Post conditions

(b[1]' = b[1] $\wedge$ b[2]' = b[2] $\wedge$ b[3]' = b[3] $\wedge$ b[4]' = b[4] $\wedge$ b[5]' = input_hi? $\wedge$
b[6]' = input_lo? $\wedge$ b[7]' = b[7] $\wedge$ b[8]' = b[8]) $\wedge$
Station Datatype Table' = Station Datatype Table – b $\cap$ b'

Figure 22
Z schema for the Entering The Limits module

4.3.3.3. Validating Raw Data

Validate Labels

Ξ Parse Table && a list of datatypes for each station and their position in the raw data file
Ξ Station Table && a list of station names
Ξ Raw Data Database && Raw Data Database

input_sta?	:	STATION_NAME
sta_no	:	STATION_NUMBER
a	:	STATION_TABLE_RECORD
raw_file	:	CHAR 16
raw_rec	:	RAW_DATA_RECORD
type	:	RECORD_TYPE
sta_no_pos	:	POSITION
c	:	PARSE_TABLE_RECORD

Invariants

$\exists a: (a \in station\_table \wedge a[1] = input\_sta?) \vee$ "All" = input_sta?

Preconditions

raw_file = input_sta? +".dat"

$(\exists raw\_file: raw\_file \notin raw\_data) \Rightarrow$ (msg! = "File " + input_sta? +".dat does not exist")

$((\exists raw\_file: raw\_file \in raw\_data) \Rightarrow$
$((\forall raw\_rec: raw\_rec \in raw\_file \wedge type = raw\_rec[1]) \wedge$
$(\exists a: a \in station\_table \wedge a[1] = input\_sta? \Rightarrow sta\_no = a[2]) \wedge$
$(\exists c: c \in parse\_table) \wedge c[1] = sta\_no$
$\wedge c[2] = type$
$\wedge b[4] =$ "Station_number" $\Rightarrow sta\_no\_pos = b[3]) \wedge$
$(raw\_rec[sta\_no\_pos] \neq sta\_no) \Rightarrow$
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field " + raw_rec[station_no_position] + "."
$) \wedge$
$(\forall raw\_rec: raw\_rec[2] \neq YEAR(DATE()-1))) \Rightarrow$
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field 2 which is the year field."
$) \wedge$
$(\forall c: raw\_rec[3] \neq JULIAN(DATE()-1))) \Rightarrow$
msg! = "There is an error in station " + input_sta? + " at record " + raw_rec +" in field 3 which is the Julian date field."
$))$

Post conditions

Figure 23
Z schema for the Validating Raw Data module

4.3.3.4. Validating Data For Ever

Validate Measurements

Ξ Station Table && a list of station names
Ξ Station Datatype Table && a list of datatypes for each station and their position in the raw data file.

Ξ Measurement Table

input_sta?	:	STATION_NAME
a	:	STATION_TABLE_RECORD
input_type?	:	DATATYPE
b	:	STATION_DATATYPE_TABLE_RECORD
begin_date?	:	MEASUREMENT_DATE
end_date?	:	MEASUREMENT_DATE
da	:	MEASUREMENT_DATE
time	:	MEASUREMENT_TIME
hi	:	WARNING_UPPER_BOUND
lo	:	WARNING_LOWER_BOUND
measurement	:	MEASUREMENT_VALUE
c		MEASUREMENT_TABLE_RECORD

Invariants

$\exists a: (a \in station\_table \wedge a[1] = input\_sta?) \vee$ "All" = input_sta?

$\exists b: b \in station\_datatype\_table \wedge b[1] = input\_sta? \wedge b[2] = input\_type?$

Preconditions

$((\forall c: c \in measurement\_table \wedge begin\_date? <= c[3] <= end\_date? \wedge c[1] =$
input_sta? $\wedge$ c[2] = input_type? $\wedge$ da = c[3] $\wedge$ hr = c[4]
$) \Rightarrow$
$((\exists b: b \in station\_datatype\_table \wedge b[1] = input\_sta? \wedge b[2] = input\_type?$
$hi = b[5] \wedge lo = b[6]$
$) \Rightarrow$
$((c[5] >= hi \vee c[5] <= lo) \Rightarrow$
msg! = "There is an error in station " + input_sta? + " for datatype " + input_type? + " on " + da + " at " + time + "in which the measurement is out of range."
)))

Post conditions

Figure 24
Z schema for the Validating Data For Ever module

Chapter 5

5. Conclusions

This project is an up and running project. It has been running for several years. During this time the system has run reasonably well and appears to be fairly robust. It is reasonable to believe that the project is technically and logically sound. This of course is never the case. There is always some little window for escape so that if we really try we can snatch defeat from the jaws of victory.

With the use of formal methods, the reader can hopefully see some light at the end of the tunnel. You should be able to see that it is at least possible to write definition and/or design using formal methods. In real life this is still a hard sell. For this project, which has been ongoing for several years, there is little or no design, not even a hand written flow chart. The peculiar part is that the only chart they do have is their database schema which is a form of a relational graph. This graph shows the relationship among several data elements and is one of the newer forms of informal methods.

5.1. Work at the ENP Research Center

As I have mentioned before, a system with the magnitude of the ENP hydrology system can always use a few enhancements. I will say more about in chapter 7 "Future work". This means that there will always be a need for someone working on both the hardware and software. This has been going on for years and, so far, nobody has made a big mess. This is especially lucky because they not only have few methods of

development, they have few methods of operation. I have not as yet seen a Policy and Procedure Manuel. I have not seen a software review board. It is nice to be able to operate without all these rules, but it is a little risky, especially if there are several people working on the system. With this in mind, it is no wonder that they have few development methods and that none of them are formal.

5.1.1. The Good

It works. Anything that works is good. If it works, don't fix it. The first two statements are true and the third one would be true if it were not an ongoing expanding development. This is a research project and each time a question is answered, two more questions arise. I expect this project to last for decades and have enhancements added to it for the rest of it's life. It is fine to go along as they have for the past decade, but it would be better in the long run to use the latest state of the art technology and latest methods of development. The current state of the art is to have standard methods for developing software. The latest state of the art is to use formal methods. They have started to write the documentation for Data For Ever and they have written a Database Schema for Data For Ever. The database schema seems to follow the philosophy "one to many" relations. They have also started the "Data For Ever Documentation".

5.1.2. The Bad

Every project has a few problems that show up from time to time. I will mention a few here, but each time anyone examines a piece of software, they will probably find a few of these. Some of these are minor and not worth fixing and some are very small but the fix would be too complicated to make it worth while.

The documentation that they have has a good start, but, like many projects, documentation is given a low priority and completed last. Of course there is no last because this project is a research project and will be in development as long as it exists. This means that if you put off writing the documentation until the project is complete, it will never be written. When the system is enhanced, there is no requirement that the documentation be updated. If the enhancement had to pass a software review board, documentation could be a requirement. Other requirements could be specified as well, such as proof that the new module worked properly, the new module did what the users wanted instead of what the developers want, etc.

There are several names that are similar or the same. The names Station Table, Datatype Table, and Station Datatype Table can be confusing if you are not careful. The field name "station" is used in several tables and it means a different thing in different tables. For instance: in the Station Table, the table name is "station", and the field named "station" means the station name. In the parse table "station" means station number, and in the measurement table "station" means station name again. There are not many of these problems, but it is not a good idea to have a name that can mean two or more different things.

In Data For Ever we have a datatype of datatype. In most software systems, the word datatype refers to the type of data that a variable is composed of, i.e. CHAR, INTEGER, FLOAT, etc. In Data For Ever the developers have used the word datatype to describe the types of measurements. Therefore, if you are measuring the battery voltage, you have a datatype of "Battery voltage" which has a datatype of CHAR 25.

For the raw data, the data is put into a buffer that is 256 characters long. The assumption is that the maximum size of Raw Data Record will be 256 characters. This is fine as long as everything is working properly, but it may not be so fine if things begin to fail. All sorts of problems, such as record overrun, no terminating carriage return, etc. can create undetectable or unmanageable errors.

The measurement table is incomplete. It has no date, time or person for the Estimation Event. It also has no time for the Validation Event.

5.1.1. The Ugly

Some problems are more serious and should be examined under a much brighter light. I have found a few of these, but there are probably more demons in the background. A few of the glaring problems are listed in the following paragraphs.

I have mentioned the discrepancies in the documentation from the standpoint of the permanent staff members. If you look at this matter from the standpoint of newcomers the opinion would probably be a little more critical. The documentation is somewhere between non-existent and terrible. The people that are working on this project are long-term career people. If anyone has been working on a project for years, there is little or no need for documentation. It should be understood that it is very costly and time consuming to build and maintain documentation. It is the full time staff that decides what is to be completed in what order, so if they don't need this documentation for themselves, the common attitude is "why not let the newcomers figure it out themselves".

When you check the raw data, nothing is sacred. You cannot rely on anything. There is no way to tell if anything is accurate. This creates a great dilemma:

> You have several pieces of information coming over the airways. You have requested information to be sent from a monitoring station. The dilemma is "how are you going to insure that the data is correct?". You can be pretty sure but never positive. You cannot check the record type unless you check the measurements first. There is nothing to test it against. You can check it against the possible valid record types in the Parse Table, but what if it is in error and is the same as another valid record type with the same number of measurements. This is a very small if, but it is there. If you cannot check the record type for positive accuracy, you cannot check the measurements.

5.2 Work on formal methods

When you use formal methods, everything has to be exact. You cannot write script if there is anything missing. When it was time to implement the automatic validation, it was found that it could not be done without adding three fields to the database. There could be more added, and I will discuss this further in Chapter 7, Future Work.

You may have noticed that checking the station number requires the use of three tables, i.e. the Station Table, the Station Datatype Table, and the Parse table. This is a very cumbersome process and could be accomplished with only one table with a few alterations. I will discuss this further in Chapter 7, Future Work.

Another problem with informal methods is that there is no check of datatypes. There is no way to write a specification that insures that you are using the proper datatype. With formal methods, everything is declared and you cannot use the improper datatype.

Chapter 6

6. Significance

The significance of this study lies in two areas. The first area is to demonstrate the use of formal methods in software engineering. The second area is to produce a solution to the problems encountered by the USGS-BRD at the ENP. The questions then become: Who cares? Why do they care? What is this study for? Who does this study help? Why is it so hard to sell?

Who cares? The software developers care. The software users care. The software maintenance group care.

Why do the developers care? Because if the definition or the high level design is done in formal methods, it is more exact. It has to be mathematically correct and there is no straying from the specifications. This makes the decision making process of implementation more straightforward. It also leaves the developer a rock solid backup for any questions as to the functionality of the final product. Once formal methods are perfected and understood by the majority of software developers, there will be no way to stray from the design. The developer will be able to track the progress of the development of the product from the time of definition to the time of test and delivery.

Why do users care? Because if they order a product and it is designed formally, there is a very much higher likelihood that it will work. Once software engineers are familiar with the use of formal methods, they will find that it is just as fast (or faster) to

design software formally as informally. With the promise of more accurate result, why use anything else?

Why does the maintenance group care? Because when additions or changes are needed to the system, it is easier to find out exactly how the system works and what would be the proper fix.

Why do they care? The software users care because it is cheaper, faster, and more correct. When anything is designed in mathematical form it is bound to be more exact. If a product is designed more accurately the first time, there will be less rework and less backtracking from the coding and testing stages to the design stage. This makes development more efficient, as it is less costly and faster.

What is this study for? It is to demonstrate that formal methods can and do work in real life. There is a place for formal methods of operation and the sooner software groups recognize this the better off they will be.

Who does this study help? This study helps everyone who is developing software of any magnitude. The person who is writing a few lines of code to produce a simple program may not benefit from formal methods at first. They should be very careful, though, because if their competition is using formal methods, they will soon be left in the dust.

Why is it so hard to sell? It is a hard sell because it is new. Anything new is susceptible to skepticism. I have no doubt that this path is inevitable.

Chapter 7

7. Future Work

The work at ENP is a never ending job and there will always be more work to be performed on the software systems. I can only mention a few that seem eminent.

As for formal methods, there is an end but it is not in sight. Before formal methods are perfected, the operators must be simple and straight-forward. There will have to be a lot of education to get the general programming population up to speed on the use of logical arguments to prove that a project does what you want it to do. Z is one of the simplest forms of formal methods. The better methods are even more complicated than Z.

7.1 Work at the ENP Research Center

Some of the projects that are on the horizon at ENP are:

1/. Validating Archival Data

Once the validation modules get running, there will be other areas for its use besides validating the daily input. There is a lot of old data that has been around and these automatic validating tools may discover some errors.

2/. Time Series Reporting

After the regular reporting has been ironed out, It will be time to set up special reporting that happens on an irregular basis. These irregular events would be events such as hurricanes, full moon, drought, flood, etc.

3/. Automatic Estimation

The first big area of automation is automatic validation. The second big area is automatic estimation. Estimation is a very large area, and can be as extensive as your mind may go. To estimate values that are missing or questionable, you can perform almost any task from simple eyeball estimates to averages taken from the past years and from neighboring stations.

4/. Run after entering "Reason Value Missing"

Estimation can also be modified by including the Reason Value Missing in the processing logic. You may come up with a completely different answer if the Reason Value Missing is a hurricane or a dead battery.

5/. Run after entering "Validation Method"

The estimation method will also be affected by the method with which the data was validated.

6/. Changes to the database

I have recommended a few changes to the database. It should be noted that these changes are small and unnecessary unless these automatic validation methods are implemented. The changes that I recommend are

Consolidate the Parse Table and the Station Datatype Table

Add the last measured date to this consolidated table. This would be another field that saves the last measured date. This would allow the checking of the first measurements time stamp in the next day's measurements.

7.2 Work on formal methods

I have suggested that formal methods are on the rise. There are still a lot of improvements to be added to the current systems and there are fancier systems being developed every day. In Z there are several more operators that can be implemented, but they are so complicated that I left them out to encourage more people to give this method a try.

To date there are only a few software tools available to support formal methods. Software tools can be used to read the specifications and assure that they are correct. These processors would be similar to a pre-compiler. They could also incorporate reading of the code. Each time a piece of code is executed, the state of the system changes. There are a few of these processors that have been attempted, but they are experimental and none of them are marketable.

Z specifications do not consider time. There is no specification that says which process has to operate first. Some of the more complicated formal methods like Petri Nets require that one process has to finish before the next can start. This is much better than Z but I think we should get the basic model of formal methods off the ground before we try to cover all bases. Another step forward will be temporal logic which has time oriented operators built into it.

When we get the formal specifications integrating into the code, we will have the problem solved. This includes: 1/. Generating a group of logical statements that state the condition of the environment before the process is run. 2/. Inserting logical statements in the code that depict the change in the state of the environment as the

program executes . 3/. End with a group of logical statements that state the condition of the environment after the process is run. In all of this it would be nice to add some timing constraints to insure that no process tried to run before it was ready. The first step is discussed in this thesis and the rest are up for grabs.

LIST OF REFERENCES

1. A Specifier's Introduction to Formal Methods
   Jeanette M. Wing
   Computer TK 7885.A1 C35

2. Applications of Formal Methods: Developing Virtuoso Software
   Susan L. Gearhart
   IEEE Software QA 76.75 I33

3. Seven Myths of Formal Methods
   Anthony Hall
   IEEE Software QA 76.75 I33

4. Specifying a Real-Time Kernel
   J. Michael Spivey
   IEEE Software QA 76.75 I33

5. Domain Specific Architecture Development for Enterprise Systems Based on Common Object Request Broker Architecture
   Vidya G. Bhat
   QA 75.499.B34 1999

6. Understanding Z
   J. M. Spivey

7. The Unified Modeling Language Users Guide
   Grady Booch, James Rumbaugh & Ivar Jacobson

The End