LOAD MANAGEMENT CONTROLS FOR SURFACE IRRIGATION

by

Michael T. Lasch

B.S., Michigan State University, 1985

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Agricultural Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

Harry L Manges
Major Professor

LD
2668
.T4
AGE
1987
L37
c. 2

A11207 308882

ACKNOWLEDGEMENTS

This work is dedicated to my family which provided unfailing support in my endeavors.

I would like to express my gratitude to Dr. Harry Manges, without whose guidance and encouragement this project would have never been completed. Thanks are extended to Steve Young and Doris Grosh for serving on my committee. Their constructive criticisms have been incorporated throughout my work.

My thanks are also extended to Phil Barnes and the staff of the Silver Lake Experiment Station for providing a platform for my research.

For the funding of this project, I thank the Agricultural Experiment Station and Kaw Valley Electric.

I would like to express my gratitude to the faculty, staff, and fellow graduate students that made my work in Manhattan pleasant. Mike Schwarz and Sheri Shanks deserve special mention. My gratitude extends to Anne Moore for her hours of editing.

TABLE OF CONTENTS

Page

INTRODUCTION..... 1

LITERATURE REVIEW..... 3

PROBLEM DESCRIPTION..... 7

SYSTEM CONSIDERATIONS..... 15

MATERIALS AND EQUIPMENT..... 17

Field Hardware..... 17

Computer Description..... 17

Carrier Current System..... 21

SOFTWARE CONSIDERATIONS..... 25

PROGRAM DESCRIPTION..... 27

DESCRIPTION OF INTERFACE BOARD..... 32

RESULTS AND DISCUSSION..... 38

System Operation..... 38

Relationship between Temperature and Load..... 40

SUMMARY..... 46

SUGGESTIONS FOR FURTHER RESEARCH..... 49

Incorporation of Advance Detection..... 49

Remote Communication with the Controller..... 49

On-Field Scheduling..... 50

Multi-Tasking Operating System..... 50

Connection to Kaw Valley Electric..... 50

REFERENCES..... 51

APPENDIX A PORT FUNCTIONS ON AN IBM PC COMPUTER..... 53

APPENDIX B ADVANCE DETECTION..... 55

APPENDIX C SUMMARY OF CONTROL PROGRAM FUNCTIONS..... 69

APPENDIX D CONTROL PROGRAM FUNCTIONS LISTING..... 81

TABLE OF FIGURES

		Page
FIGURE 1.	Load Distribution with no Load Management....	8
FIGURE 2.	Load Distribution with Load Management.....	9
FIGURE 3.	Income Potential Lost Due to Over-Control of Loads.....	14
FIGURE 4.	Schematic of the Field and Test System.....	18
FIGURE 5A.	Schematic of the Bladder Valve in the Open Position.....	19
FIGURE 5B.	Schematic of the Bladder Valve in the Closed Position.....	19
FIGURE 6.	Schematic of the Carrier Current System.....	22
FIGURE 7.	Surge Controller Algoritnm.....	28
FIGURE 8.	Board Layout - Surge Controller.....	33
FIGURE 9.	TTL Voltage Levels.....	34
FIGURE 10A.	556 Set Up as a One-Shot.....	36
FIGURE 10B.	Function of a One-Shot.....	36
FIGURE 11.	Dramatic Increase in Demand with no Load Management.....	41
FIGURE 12.	Constrained Increase in Demand Due to Load Management.....	42
FIGURE 13.	Electric Load Dependence on Temperature.....	43
FIGURE 14.	Electric Load Dependence on Temperature Kiro Substation.....	44
FIGURE 15.	Theory of Operation - Advance Detector.....	57
FIGURE 16.	Constant Current Source.....	62
FIGURE 17.	Sensor Schematic.....	65

INTRODUCTION

Surface irrigation typically suffers from low efficiency in water usage. Efficiency refers to the the amount of water applied to the field compared to the amount of water that moves into the root zone of the crop. A solution to the problem of low efficiency is to alter the method of water application to the field.

The typical method of surface irrigation is to apply water to a set of furrows for an amount of time that fits the farmer's work schedule. This usually results in over-application of water. Previously, costs of pumping did little to induce the farmer to conserve water or energy.

Recently, increased energy prices and decreased grain prices have prompted farmers to search for alternatives to reduce costs.

One alternative is the implementation of surge irrigation. Surge irrigation is the intermittent application of water to land. The resulting flow is in the form of surges. Surge offers reduction in water advance times and a reduced variation in the infiltration opportunity time over the field (Podmore et al., 1983). The primary drawback to surge irrigation is the regulation of intermittent water application. The repetitive nature of controlling surge lends itself to automation.

In an effort to reduce pumping costs, many farmers served by the Kaw Valley Electric Cooperative near Topeka, Kansas, participated in a load management program. Rate reductions applied only if the farmer subjected his irrigation pumps to power interruption.

Load management refers to the process of temporarily interrupting service to a customer in an effort to reduce the utility's service demand. If a farmer participated in the load management program in 1986, his connected horsepower charge was cut in half. Due to the success of Kaw Valley's load management program, the connect charge has been eliminated for irrigators for the summer of 1987.

Kaw Valley implements load management around the clock as needed. The random power interruption does interfere with a farmer's irrigation schedule, regardless of the method of irrigation.

The emphasis of this research was to design an irrigation controller that would insure that the desired quantity of water was applied to a field regardless of the incidence of power interruption. The controller allows the farmer to substitute technology for his presence on the field. The controller is not intended as a substitute for management skills, but rather allows the farmer to make better use of his time.

LITERATURE REVIEW

Considerable work has been done on comparing the efficiencies of surge and continuous irrigation. Continuous irrigation is a term that applies to the continuous application of water to a field until the desired infiltration is achieved. Izuno and Podmore (1984) made extensive comparisons of application efficiency between surge and continuous irrigation. They varied the timing of water application, soil types, and length of fields. Their studies suggested that surge irrigation resulted in a more uniform depth of irrigation, since there is less variation in the opportunity for infiltration. Izuno and Podmore (1984) constructed empirical equations based on soil properties, slope of the field, and length of run to determine surge timing.

Application efficiency may be improved if cutback irrigation is instituted after the water crosses the field, Cutback involves reducing the inflow into the furrows. This reduces the amount of water that runs off the field, but maintains the potential for infiltration. Cutback irrigation capabilities were omitted from the initial design of the controller.

Heerman et al. (1983) designed an integrated irrigation control system which combined pump control, irrigation scheduling, and load control to reduce peak electrical demand. The system controls irrigation for fourteen center

pivots. As a management aid, the system estimated soil moisture based on: weather, crop, possibility of service interruption, and soil properties. The farmer used estimated soil moisture as a basis for determining the order in which his fields were to be irrigated. Power to the pumps was dropped in reverse order of the famer's priority. The software used to implement their system is described by Buchleiter et al. (1983).

There are several differences between the circumstances in which the system of Heerman et al. and this controller operate. The loads in their system were all on the same load management address. The farmer was requested to drop enough of his load to satisfy the utility, but he was not obligated to drop more than one-half of his total load. This gave the farmer flexibility to irrigate a particular field by changing his field priority schedule. In Kaw Valley's service area, an irrigation pump may not have the same address as a nearby pump. Therefore, the farmers were not able to choose which pump would be interrupted. If a pump's power was to be interrupted, the farmer had no opportunity to irrigate the field until the power was restored.

Another significant difference was that the system of Heerman et al. scheduled center pivots. When power to a center pivot is interrupted, the pivot stops at its current position. When power is restored, the center pivot continues as if the interruption did not occur. The depth of

water applied by a center pivot is controlled mechanically; it is not time dependent. A surge irrigation system must save its schedule and resume at the point of its interruption. The controller must be sensitive to the timed schedule but independent of the length of the power interruption.

Sauer and James (1985) developed a program to modify daily irrigation schedules which reduced and smoothed power requirements on a large farm using center pivots. The program shifted irrigation schedules to avoid operating during peak load conditions. The local utility's rate structure was such that their system reduced power bills by nearly 10% without affecting yields.

Automation of irrigation has not been limited to center pivot systems. Podmore et al. (1983) conducted tests to evaluate surge irrigation as applied to furrow irrigated corn. To decrease the amount of labor required to conduct surge irrigation, the investigators developed automated techniques to control the water flow. Pneumatically operated pillow valves were used to regulate the flow of water. The pillow valves required a source of compressed air at the field.

Weckler et al. (1984) determined that surge irrigation could be used to reduce furrow run-off. A microprocessor-based controller was developed to implement the surge schedule. The controller was based on the Z-80 and

programmed in assembly language. Pneumatically driven valves were used to control the application of the water.

The possibility of eliminating pneumatically controlled valves was demonstrated by Bradbury and Manges (1984). They constructed a surge irrigation system which used wire telemetry to control bladder valves. Bladder valves make use of the kinetic energy of the water in the pipe to switch the valves. The state of the valve was controlled by a solenoid. The need for a compressed air source on the field was eliminated. Their system was controlled by a Synertek SYM-1 single board computer and was later upgraded to a Commodore VIC-20. This design did not account for the possibility of load management conditions.

PROBLEM DESCRIPTION

Kaw Valley purchases all of its power from Kansas Power and Light. Kaw Valley's purchase agreement requires substantial penalties for establishing a new demand peak. Kaw Valley is charged eighty percent of their peak demand each month. This amount is in addition to the charge for the actual power consumed. The peak charges are used by the generating utility to offset the capital costs incurred in establishing facilities. This means it is not locked into peak charges established several years earlier. The possibility of reducing the demand peak charge gives Kaw Valley incentive to implement load management.

If their peak demand is higher than the average consumption, significant savings could be realized by reducing the peaks. In 1985, Kaw Valley Electric implemented a load management program in an effort to reduce costs. In the 1986 calendar year, Kaw Valley saved $204,000 (Winnerling, 1986) as a result of load management.

Load management provides an opportunity to shift the service demand to a different part of the day. In this instance, irrigation pumps are taken out of service during peak demand periods (afternoon) and are returned to service during periods of less demand (nighttime). The shifting of demand is apparent by comparing Figures 1 and 2. No load management was instituted on the day shown in Figure 1.

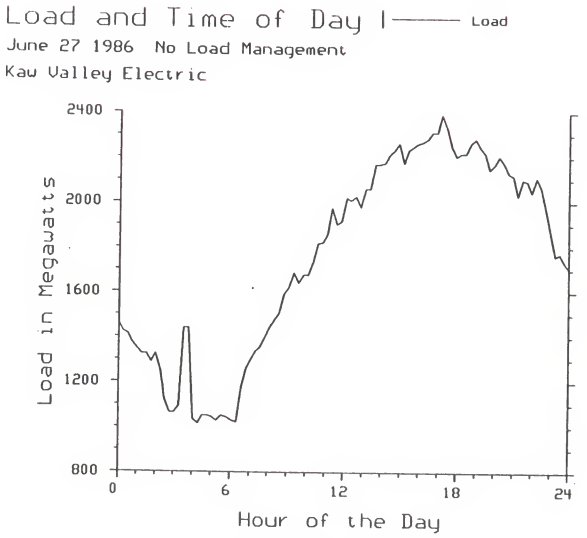


FIGURE 1. Load Distribution with no Load Management.

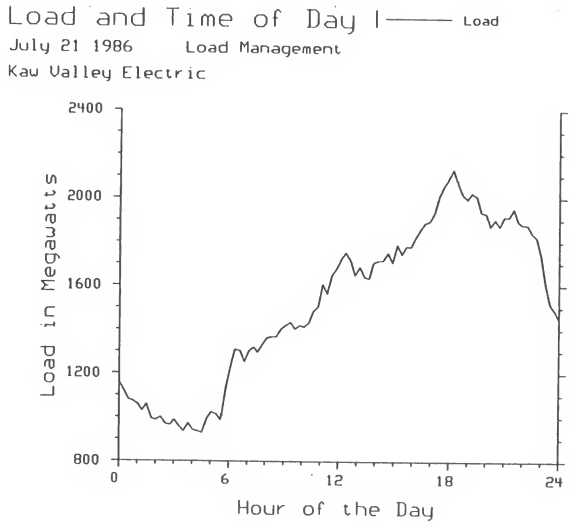


FIGURE 2. Load Distribution with Load Management.

Figure 2 shows a day in which load management was implemented from about noon to until midnight. The increases in demand are less dramatic in Figure 2, since load management was used to reduce service demand.

The ratio of average load to peak load is a utility's load factor. Before load control was instituted, Kaw Valley had a load factor of 39 %. In 1985, the load factor was increased to 47 %. In the first year of operation, load management was instituted only in the summer. Although, Kaw Valley monitors peak loads during all seasons, summer weather had the potential to cause peak conditions due to irrigation and cooling loads.

The different types of loads managed on Kaw Valley's grid were: irrigation pumps, residential air conditioners and water heaters, a radio station, and commercial customer air conditioners. Kaw Valley divided its service area into seven zones. The loads for each zone are managed independently. Each zone is fed from a different point on KPL's grid. Two zones serve all the irrigation pumps on the Kaw Valley system.

Kaw Valley installed a Westinghouse EMETCOM load management system. The system was capable of providing one-way and two-way communication with the central office. The system monitors the substation loads every fifteen minutes. The two way communication is used to monitor loads at commercial sites and to perform remote metering; however,

the expense of the two-way communication devices prevents their installation at all load management sites.

The EMETCOM system is controlled by a MicrovaxII minicomputer. The Microvax communicates with smart controllers at each substation via a microwave link. The controllers perform load calculations and switch capacitor banks into service as needed. The smart controllers reduce the load on the Microvax. The status of the substation is transmitted to the central office where the load is evaluated. If a substation warrants load management, the Microvax issues a command to drop some loads. The loads on a substation are split into several addresses which allows the cooperative to shave its load incrementally.

Irrigation pumps are good targets for load management. A typical irrigation pump in Kaw Valley's service area consumes the same amount of power as nearly sixty home air conditioners. In 1986, thirty eight irrigation pumps were included in the utility's load management program. Thus, irrigation pumps are good targets for load management since they represent a large load and a small customer base.

Crops are insensitive to the time of water application. The same cannot be said of air conditioning demands. By interrupting service to irrigators during the day, the cooperative decreases peak demand and shifts irrigation demand to a time when the total demand is low. Due to the cooperative's purchase agreement, the managers would like to

have the demand at 80% of peak conditions, since they pay for their peak consumption. Ideally, the cooperative would smooth their power demand without setting new peaks.

However, irrigation loads can be interrupted only to a limited extent. The time required to irrigate a field is dependent on the size of the field, the capacity of the well, the capacity of the soil, and the efficiency of the water application. If a farmer's yield will be affected because he is denied the opportunity to irrigate his crop, the farmer is less likely to participate in a load management program. The cooperative then loses the capability to manage his power demands.

The performance of irrigation pumps was measured in the Kaw Valley service area by Black and Barnes (1986). The average performance of the measured wells was fifty seven percent of the Nebraska pumping plant performance criteria. Improving the well efficiency would make the farmer's crop less susceptible to load management, since less time would be required to apply the same amount of water.

Load management has prompted farmers to monitor the soil moisture level more closely. The occurrence of load management increased the number of calendar days required to irrigate a field. If the soil moisture levels fell too low and load management occurred, the yield would suffer because of water stress. Barnes et al. (1986) demonstrated that irrigation scheduling by estimating soil moisture depletion

made it possible to avoid yield losses from the load management implementation.

Kaw Valley's load management system is limited by a lack of two way communication between the loads and the central office. Most of the loads on the Kaw Valley grid are controlled one-way devices. The central office can switch the power to the loads. Two way devices would allow the central office to monitor the status and demand of a load. Two-way devices were not installed at irrigation sites because of high initial cost.

With the use of one-way load control, Kaw Valley cannot determine the change in load resulting from a load management command. Since the change in load must be estimated, Kaw Valley over-controls the loads so a peak does not result from insufficient load management. This led to the utility realizing less than full income potential. The utility must pay for its peak demand, so it could reduce costs if it sold power to the peak. Over control of the demand occurred on July 21, 1986 (Figure 3). The peak demand for the day was less than had been established on June 27, 1986. However, the managers would rather over-control the loads, rather than set a new peak.

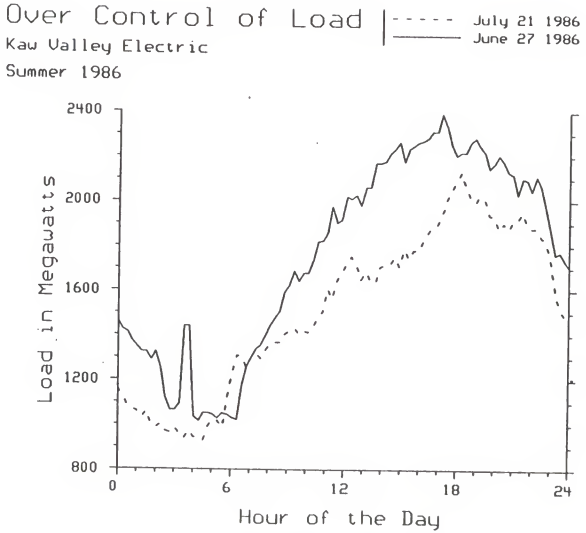


FIGURE 3. Income Potential Lost Due to Over-Control of Loads.

SYSTEM CONSIDERATIONS

Flexibility of the irrigation controller was a high priority in the design. The controller had to be capable of acting as a surge controller, as well as a continuous irrigation controller. An irrigation schedule was selected irrespective of the possibility of load management.

If load management was instituted after irrigation was started, the controller logged the power interruption. When power was restored, the controller resumed the irrigation schedule as if load management had never occurred. This insured that the desired amount of water was applied to the field. The controller continued in this fashion until the irrigation schedule was completed, then the controller toggled a relay to turn the pump off.

The controller was capable of addressing six valves. If part of the field was to be irrigated, valves could be left out of the irrigation sequence. The user programmed the amount of time that water flowed through each valve. If the user alternated the water application between two valves, surge irrigation would be implemented. The program allowed a valve to be actuated up to twelve times.

Flow through a valve could be from minutes to days. A very long set time would in effect be continuous irrigation. The operating sequence of the valves was at the user's discretion; the controller did not necessarily cause valves one

and two to be actuated before valves three and four. The controller did not force the operator to open valves that were side by side.

A prime consideration was that the controller be easy to program. The controller prompted the user for information, and was designed to then operate without further user assistance; the user could leave the field. As mentioned previously, the controller cut power to the pump when the schedule was completed. However, a facility to interrupt the progress of the irrigation sequence was included. This made it possible to stop the program if conditions warranted. A suspend command was included for user convenience. The suspend command differed from the stop command in that the schedule could be resumed after the suspend command was lifted. It was not possible to alter the irrigation schedule without re-starting the program.

The controller provided status information about the irrigation schedule being executed. The current time was displayed to show the user that the controller was functioning. The current stage of execution was displayed along with the length of the current cycle and elapsed time that water had flowed during that cycle. The status of load management was displayed so the user could see if the utility had taken his pump off-line.

MATERIALS AND EQUIPMENT

Field Hardware

The controller was tested on Sam Kelsey's farm near Silver Lake, Kansas. The field measured 1550 ft. x 1300 ft., approximately one-third the size of the typical surface irrigated field. The furrows ran 1300 ft. and furrow spacing was thirty inches. The crop on the field was corn. Eight inch gated pipe was used to apply water to the field. A schematic of the field is shown in Figure 4

The controller actuated six Hastings bladder valves. The bladder valves (Figures 5a and 5b) were controlled by a solenoid in a pitot tube line. When the solenoid allowed flow through the pitot tube, the bladder inflated and cut off flow through the valve. When the solenoid switched, it blocked the pitot tube and allowed the bladder to vent to the atmosphere which caused the bladder to collapse and let water flow through the valve body.

The pump at the field delivered 890 gallons per minute at a pressure at the pump of 3 pounds per square inch. It was driven by a 30 horsepower, three-phase motor.

Computer Description

The choice of the computer to serve as the controller was based on many factors. The majority of the development time was to be spent programming the computer, not attempt-

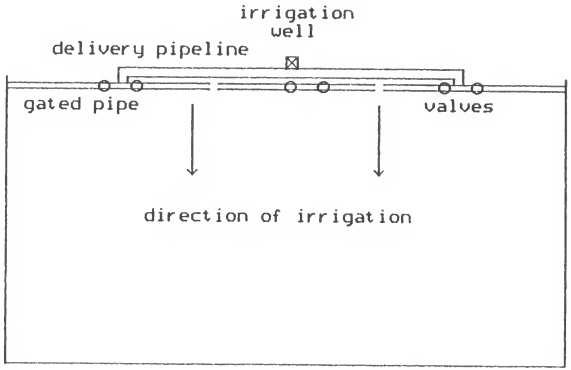


Figure 4. Schematic of the Field and Test System.

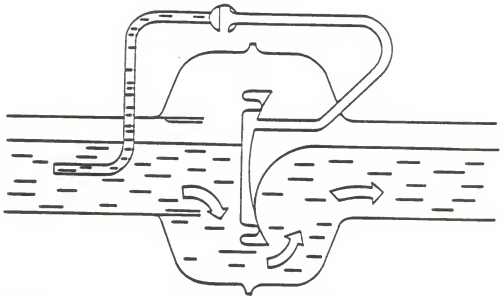


FIGURE 5A. Schematic of the Bladder Valve in the Open Position.

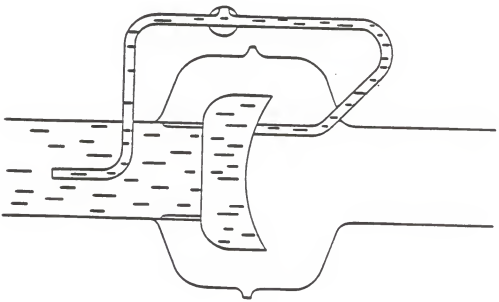


FIGURE 5B. Schematic of the Bladder Valve in the Closed Position.

ing to bypass shortcomings of the computer system. It was desirable to use a computer that was widely available since this would improve the likelihood of obtaining service if required. It also meant that a large group of knowledgeable users existed.

A full keyboard, monitor, and floppy disk drive were considered vital components of the computer system. Other hardware requirements were at least one empty ROM (read only memory) socket or slot on the processor board and a parallel printer port. An empty socket would allow a custom ROM to control the computer. The parallel port would be used to connect the valve control devices. A parallel printer port would use a simpler interface board than a serial port. A parallel port has several lines that transmit data simultaneously. If few enough devices were to be controlled, a line could be assigned to a single device. Serial ports transmit nearly all data through a single line.

IBM PC compatibles support the desirable features. They have a large user base. MS-DOS provides a stable operating system. The IBM PC standard supports up to 640k RAM, providing a large workspace for program development. Many languages are supported by the IBM PC class machines.

An IBM PC compatible computer was used to implement the controller. The computer consisted of: a motherboard containing an IBM PC compatible basic input output system (BIOS), 150 watt power supply, a composite monitor, a

standard keyboard, a floppy drive, and an input/output card. The input/output card contained a serial port, a parallel port, and a real time clock.

The motherboard had eight expansion slots. The central processor was an 8088 running at 4.77 Megahertz. The motherboard also had empty sockets for an 8087 numeric coprocessor and four additional ROMs. A color graphics adapter was used to maintain compatibility with computers in the Agricultural Engineering Department. Additional equipment included a 10 megabyte Winchester hard disk and an Adaptec hard disk controller. Since the hard disk was hermetically sealed, the field conditions did not affect the operation of the disk drive.

Carrier Current System

A carrier current link was used to transmit actuating signals to the bladder valves. A schematic of the system is shown in Figure 6. A carrier current system superimposes a high frequency signal on the same lines that supply power to the device being controlled (Stearne, 1981). The carrier current hardware was the same as used by Bradbury and Manges (1984) in their surge controller. They used a 24 volt carrier current system to reduce shock hazard.

A carrier current system is a form of hard-wired radio transmission. Instead of the signal being broadcast through the air, it is propagated on the power lines. This allows

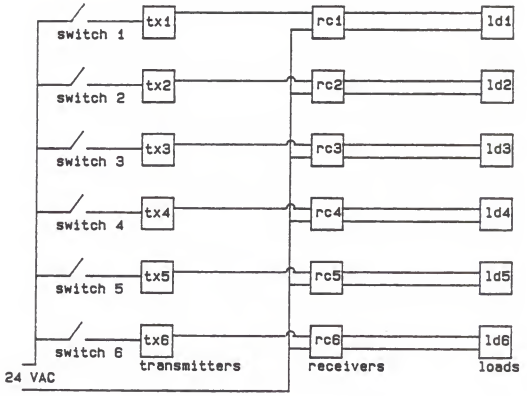


FIGURE 6. Schematic of the Carrier Current System.

signals to be transmitted without the complexity of a radio link. However, there can be problems, since AC lines were not intended to carry the high frequency signals typically used in carrier current systems. There can also be problems with coupling a high quality signal to an AC line. The coupling device must isolate the power supply voltage of the transmitter or the receiver. The coupling of the transmitter/receiver pair to the power lines is usually accomplished by means of a capacitive circuit. Carrier current systems may be degraded by the noise generated by motors and other electrically noisy devices.

The bladder valves had two desired states, opened or closed. There was no feedback on the state of the valve. Therefore, if a valve failed to change state, the controller could not detect it. No workable means of providing feedback was apparent. The valves took as long as a minute to change state. If the solenoid changed state, the valve did not necessarily follow suit. If there was insufficient velocity through the valve body, the valve would not close regardless of the solenoid position. Positive valve closure probably could have been detected by mounting several pressure switches within the valve body. However, all the switches would need to be activated at once, since the valve would slap if it were not closed. A decision was made to forego the feedback loop. The lack of feedback and binary state of the valves significantly reduced the complexity of

the data link.

Each valve was assigned its own transmitter/receiver pair. All of the valve signals were transmitted on the same wire pair. This allowed the valves to be controlled without a control line dedicated to each valve. The control and power lines need only be as long as the distance between the farthest valves. The receivers were triggered by their unique signal. The signals from several transmitters could be on the wires at the same time without interfering with the different receivers.

Southwood Electronics, the manufacturer of the carrier current system, spread the frequencies out widely. The six frequencies used were: 107 kHz, 130 kHz, 167 kHz, 200 kHz, 233 kHz, and 250 kHz. These frequencies prevented the signals from interfering with each other, even when several were present at once. Each transmitter contained an oscillator which generated its unique frequency when power was available. The receiver contained active filters which closed a relay when its frequency was detected on the line.

The carrier current system required AC voltage to operate. However, the solenoids at each valve operated on DC voltage. A rectifier was placed at each receiver to supply the necessary DC voltage.

SOFTWARE CONSIDERATIONS

Several factors were considered in the choice of the programming language. It was desired to use a well structured high level language; a well structured language aids in program development and debugging.

A useful feature of some languages is the ability to compile modules of code separately. This allows the program to be divided into separate units. If a module needs to be upgraded, the whole program does not need to be modified. This alleviates the need to recompile the whole program, resulting in a significant time savings. The separately compiled modules are then joined together by a program called a "linker."

Ease-of-use features that are important in application programming include: descriptive error messages, capability, and speed. These features are dependent upon the size of the language. Too large a language requires overhead which may hinder its speed or require a larger computer. Yet, a larger language tends to have more descriptive messages and extended capabilities. These factors must be considered in choosing a language.

Closely related to languages' ease-of-use is the time required to learn the language. Some languages are intuitively obvious, others are less so.

Another trait to be considered would be complete independence from a specific computer. This would allow the application to be transferred to nearly any convenient computer. This is a noble idea but is rarely realized. The authors of a language usually cannot resist the temptation to include machine specific features. Some languages are more easily transported from machine to machine, but none have complete independence.

Even though machine independence would be convenient, there are advantages to using a language that makes use of specific features of a computer. Support of machine specific features violates the intention of high level languages, which is to remove the user from the hardware. An example of a machine specific feature would be the real-time clock available in IBM PC class machines. It is often useful to refer to the clock, but not all computers have a real-time clock. Supporting a clock makes languages more difficult to transport across computers.

The programming language used to implement the controller was Computer Innovations C86 C. The desirable features of C include: low level hooks to the hardware, small executable programs, the capability to produce stand-alone programs, some level of transportability, and a language known to local group of programmers. It is arguable as to whether C can be called a high level language in that it supports many machine specific features.

PROGRAM DESCRIPTION

The controller program was written in C. The authors of C have published their definition of the language (Kernighan and Ritchie, 1978). This allows programmers to know the standard theory and syntax of the language.

All C programs begin execution in a "main" program. Other subprograms may be called by the main program. The thread of a program can be followed by tracing its function calls. C makes any function available to the main function or any subprogram.

The program could operate on any IBM PC compatible computer running MS-DOS with a parallel printer port and a real-time clock. The real-time clock was a battery powered clock that operated independently of the computer's power. The parallel printer port provided a means to access the board used to control the bladder valves.

The controller's main program was little more than a scheduler for the functions (Figure 7). First the main program executed the setup function which established the irrigation schedule. The function queried the user for the number of valves to be controlled. If less than the maximum (six in this case) was specified, the program skips the other valves. The user did have to enter null on-times for these valves. The term "on-time" refers to the length of time that water flows through a valve. The program then

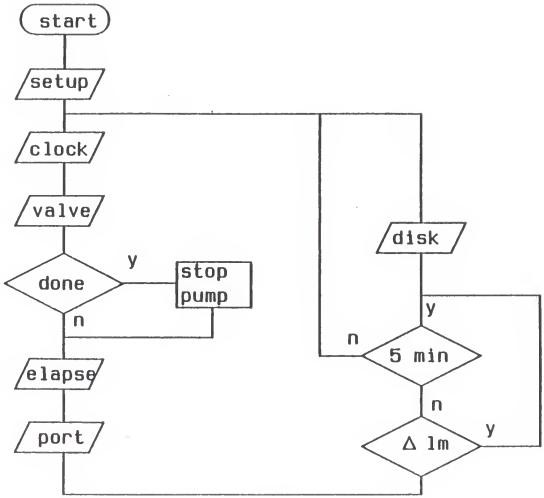


FIGURE 7. Surge Controller Alogrithm.

asked the user to enter the number of times the valve would cycle in order to complete the irrigation sequence. The function then requested the on-times for each of the cycles that were to be executed. The setup function sorted the irrigation schedule according to the order of execution that had been entered by the user. The setup function also checked for invalid entries.

After the setup function had executed, control reverted back to the main program, which then scheduled the remaining functions in an infinite loop. The infinite loop could only be broken by the operator. If the loop was interrupted, the program execution was disrupted. Each remaining function was written so that it would execute without the possibility of entering an infinite loop condition. This assured that every controlling function had an opportunity to execute.

Once the irrigation schedule was set, the program functioned without additional user input. This was an important feature since the irrigation schedule used as a test took over a day to complete.

The first function in the infinite loop was the key_act function. Key_act acted as a keyboard filter. It ignored all keyboard activity except function keys. When certain function keys were pressed, it diverted the program flow. A function key was used to abort the program. Others printed a help menu or the execution order and on-times for the irrigation schedule. After executing these functions,

control returned to key_act. Key_act then finished executing and the flow returned to the main program.

The next function called by the main function read the real-time clock and displayed the time on the monitor. Since the rest of the program depended on timing, the clock function executed before the others.

The succeeding function, next_valve, called another function (valve_exec) to determine if the elapsed time a valve was open exceeded the scheduled on-time. If the elapsed time was exceeded, valve_exec would reset the elapsed time. The called function returned a flag to signal this condition. If the flag indicated the on-time was exceeded, the execution schedule was incremented.

The elapsed time a valve was open was computed in the following function. The elapsed time variables were globally declared so that they would be available throughout the program. Incrementing the elapsed time depended on the condition of load management. If the power to the pump was interrupted, the elapsed time did not increase. The status of load management was determined by the lm_flag function. Lm_flag checked a pin on the parallel printer port which was connected, by means of a relay, to the power leads to the pump. The relay isolated the printer port from the voltage levels in the power line. If the pump was on load management, a message was printed on the screen.

The valves were controlled by the port function. The printer port had eight lines for passing characters to a printer. Six lines were dedicated to the six valves to be controlled. The seventh line was used to activate a relay which could switch the pump off. While the schedule was executing, the relay was on. When the schedule was finished, the power to the relay was cut. This in turn cut power to the pump. Since there was a line for each valve, the presence or absence of a signal was the only data required to actuate a valve.

After the port function was executed, the flow of the program returned to the key_act function.

DESCRIPTION OF INTERFACE BOARD

An interface board (Figure 8) was constructed to allow the computer to control the bladder valves and the power line to the pump. The same board detected the condition of load management. The board was needed because the parallel printer port could not directly drive the output lines, nor could it handle the voltage levels on the input side.

The addressing of the printer port is detailed in Appendix A. The printer port uses TTL logic levels (Figure 9). A high signal on the printer port had a five volt level and was limited to 64 milliamperes of current. Relays were used on the board since the carrier current system operated at 24 volts AC.

Printed circuit board relays were used to switch the power to the carrier current transmitters. These relays were used because they drew little power and were readily available. However, they required more current than the printer port drivers could provide. The relays required nearly 80 milliamperes to close. As the printer port tried to drive the relays, the voltage levels dropped so low that the relays would not close.

No immediately available TTL components could provide enough current to drive the relays. Linear integrated circuits are capable of providing several hundred milliamperes. The 555 and 556 timer chips are linear devices that can pro-

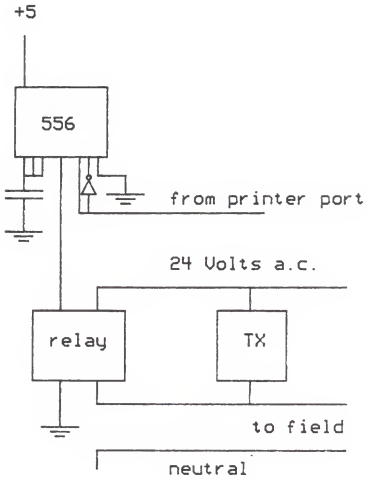


FIGURE 8. Board Layout - Surge Controller.

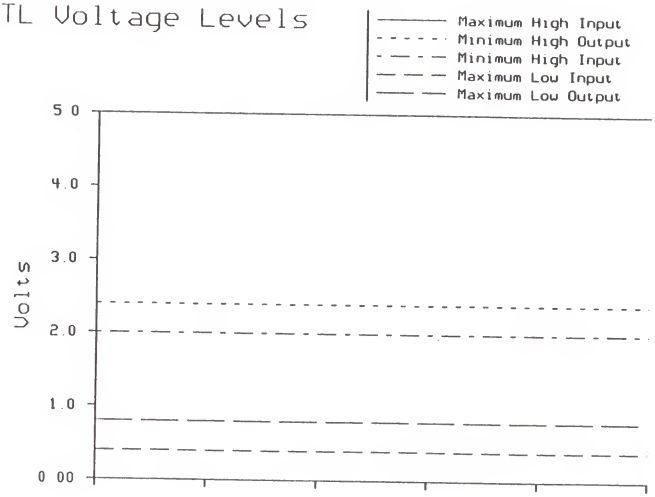


FIGURE 9. TTL Voltage Levels.

vide 200 milliamperes. The 556 is a chip that contains two 555 timers and is readily available.

A monostable circuit can be built around a 555 timer chip (Mims, 1984). A monostable circuit acts as a "one-shot" with a delayed reset. A one-shot fires once regardless of the number of triggering signals that are present. The one-shot fires on the first signal of suitable voltage. A reset signal forces the one-shot to wait for the next trigger signal. The function of a one-shot is shown in Figure 10A. Of course, with a time delay reset, the 555 could trigger again after it resets. The one-shots were reset by placing a low signal on the printer port line.

The circuit shown in Figure 10B shows the 555 in a monostable circuit except that the capacitors that allow the one-shot to reset after a time have been omitted. The omission of the capacitors prevents the circuit from resetting until another signal is sent to the chip.

The printer port could not deliver enough power to drive the 556's set up to act as one-shots. A hex inverter was used to provide the current and voltage required to trigger the 556. The hex inverters require 12 milliamps to trigger the inverters (Texas Instruments, 1985). Now that the relays could be driven reliably, the transmitters could be driven from the printer port.

The pump control line was set up in the same manner as the transmitter drivers except the the pump drew more power

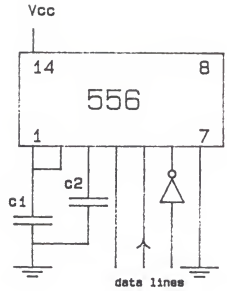


Figure 10A: 556 Set Up as a One-Shot

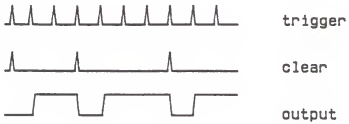


Figure 10B: Function of a One-Shot

than the printed circuit relays could handle. Two relays were staged, that is one was used to trigger the other, to alleviate this problem. The printed circuit relays were used to switch the relay rated at 250 volts and 10 amperes.

The load management status was sensed by placing a relay in the pump's power box. The relay was placed immediately after Kaw Valley's load management device but before the pump's power switch. If there was no power to energize the coil of the relay, load management had interrupted the power. The signal which ran through this relay was connected to a pin normally used to check the status of a printer. This was one of the few input lines available on the printer port. Five volts on this line indicated that load management had not interrupted the pump. The five volt level was taken from the power to the interface board.

Two different power levels were required on the interface board. The TTL levels were available from the computer itself. One of the unused floppy power cords was used to supply the five volt and ground lines. The transmitters and solenoids operated on 24 volts. One of the three-phase legs was run through a step down transformer to provide 24 volts.

RESULTS AND DISCUSSION

System Operation

There were several problems encountered during the operation of the irrigation controller. Most of these problems involved the initial operation of the system. The problem which caused the longest delay had to do with the installation of the bladder valves. There was inadequate pressure at the pitot tube to consistently inflate the bladder when the water was flowing in the main. When the valves did not fully inflate, they would slap hard enough to shake the mainline. The pressure surges caused the valves to try to push apart from the tees.

This problem had been encountered in previous work with this type of valve (Bradbury and Manges, 1984). If the operating pressure were in a different range, perhaps the problems with the valve closure would not have occurred. Constrictors were made from sheet metal. The constrictors were placed between the mainline and the pitot tube.

After this problem was solved, there were still a few incidents of the system failing. One failure was due to a solenoid sticking in the closed position. Another was due to a relay sticking. Both of these problems caused all the valves to be closed at the same time. The failures of the solenoid and relay were solved by replacement of the components.

The pressure surges due to these system failures forced the pipe apart at the tees. It also forced a plug nearly out of the gated pipe. These problems were solved by clamping the pipe and the placement of stakes in the appropriate places. Pressure surges due to the normal operation probably had the same effect, but not on such a dramatic scale.

A decision was made to install a pressure transducer at the wellhead. The transducer would stop the pump if the rated pressure was exceeded in the main. The pressure transducer was not installed before the irrigation season was cut short by rainfall events. This transducer was to be connected to an empty input line on the printer port.

A recurrent problem was encountered when the main was empty prior to an irrigation. There was not enough flow in the pipe to allow the pitot tubes to fill the bladder. This problem was solved by inflating the bladders by mouth. As stated, this would be encountered only upon the initiation of a new irrigation event. Later, it was found that filling the bladders with water before operating the system would solve this problem without exhausting the operator.

None of the other problems recurred after the system had operated for a short time. The interface board could be modified by including indicator lights tracing the signal through the board. This would minimize the time the user would have to spend finding a problem. With the inclusion of the pressure switch, there should never be damage to the

irrigation system.

The controller functioned as intended. Testing was not as extensive as planned due to above average precipitation. The controller completed two extensive irrigation schedules. As desired, periods of load management were not included in the calculation of water application times.

Relationship between Temperature and Load

In July and August, Kaw Valley began implementing load management approximately eight hours a day. The irrigation loads were interrupted near noon and the power was restored between 6 p.m. and midnight. The load profiles shown in Figures 11 and 12 show the effects of load management on the electrical demand. In Figure 11, the demand shows a dramatic rise and a quick fall after the peak was reached. Figure 12 shows the effect of load management; the demand rose quickly, then leveled as some loads were shed. The demand curve continued upward, but the effort to trim the load was evident by the many small increases.

The extent of load management seemed to be related to the temperature. The electrical load for the entire Kaw Valley service area was plotted against the daily high temperatures (Figure 13). The load followed the temperature profile. A similar plot was constructed for the Kiro substation (Figure 14); the trend is the same. The Kiro substation serves all the irrigation pumps served by Kaw Valley.

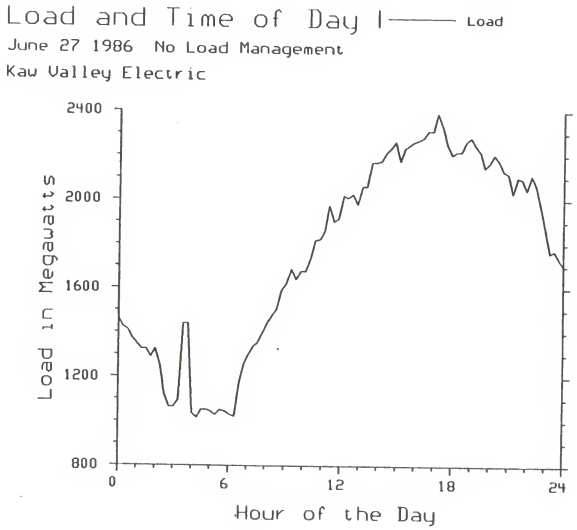


FIGURE 11. Dramatic Increase in Demand with no Load Management.

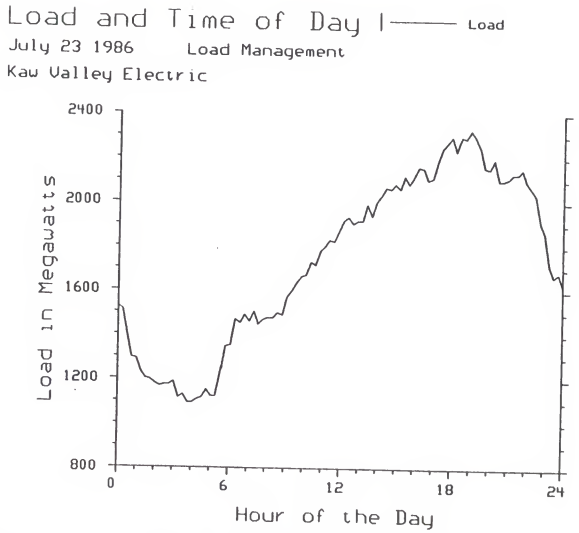


FIGURE 12. Constrained Increase in Demand Due to Load Management.

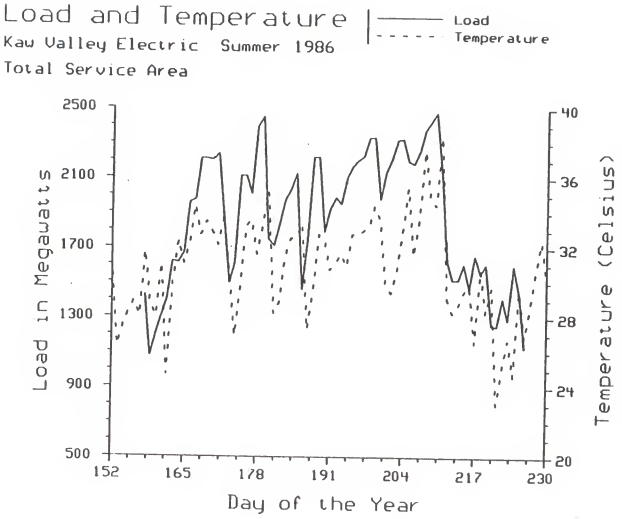


FIGURE 13. Electric Load Dependence on Temperature.

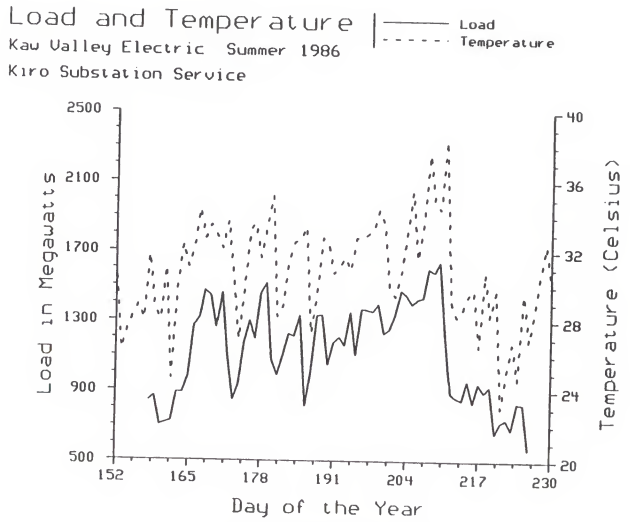


FIGURE 14. Electric Load Dependence on Temperature – Kiro Station.

Temperature could have been used to estimate the duration of load management.

SUMMARY

An irrigation controller was designed and tested to address two problems associated with surface irrigation. Surface irrigation is typically inefficient, that is not all of the water applied is available to the crop. A recent problem is the interruption of power to the irrigation pump. Electric utilities are instituting load management programs to reduce costs. Load management shifts some of the load to off-peak times. This reduces the capital expenditures associated with increasing generating capacity.

Irrigators are good targets for load management programs, as their pumps represent a significant load, yet a small customer base. Crops are not time-critical with regard to irrigation. Farmers receive financial incentive for participating in load management programs.

The low efficiency of surface irrigation occurs because a famer will schedule his irrigation around his work schedule, rather than the time required to irrigate his fields. Load management results in interruption of electric service for an indeterminate amount of time. A farmer compensates for the possibility of load management by increasing the time that his irrigation pump will run under the assumption that power will be interrupted. This reduces application efficiency and leaves the farmer uneasy about the completeness of his irrigation.

Automation of the irrigation offers a solution to these problems. This controller could be scheduled to institute surge and cutback irrigation, methods of higher application efficiency, and monitor the condition of load management. This allows the controller to apply the desired quantity of water regardless of the number or duration of service interruptions. The controller cut power to the pump when the irrigation sequence was completed.

The controller kept a running time on th length of time that the pump had power available. When load management was instituted, the controller suspended the time update, and waited for power resumption.

Bladder valves were used to control the application of water to different parts of the field. The bladder valves did not require high current to drive them as butterfly valves do. The energy to operate the valves comes from the water flowing in the delivery pipeline. The bladder valves were controlled by solenoids, a carrier current system and other hardware.

Some failures of the system occurred initially. High pressure prevented the bladder valves from closing consistently. Constriction plates were placed within the valve body to increase velocity, and thus reduce pressure at the bladders. Several relays failed, but their replacements eliminated the associated problems. Debris in the delivery pipeline became lodged in the solenoids. This prevented

their operation until they were cleaned. After the debris was flushed from the system, the solenoids performed without fail.

SUGGESTIONS FOR FURTHER RESEARCH

Incorporation of Advance Detection

A system to detect the advance of water in a furrow is described in Appendix B. Advance detection could be incorporated into the controller as a feedback loop. The controller could make use of the advance data to modify the irrigation schedule. Additionally, cutback irrigation might be implemented as the water progressed to the far end of the field.

Remote Communication with the Controller

A modem would allow the controller to be activated by a user not on the field. Of course, it would require that a phone line is available on the field. Slight software modifications would allow the user to program the controller without being at the site. A communication link would allow the user to monitor the progress of an irrigation sequence. A farmer could determine if his pumps were being load managed by the utility without visiting the site. Communications could also be used by the cooperative to implement two way communication between irrigation pumps and the central office. The utility could predict start-up loads when restoring power if their Microvax could poll irrigation pumps and determine if they were going on line.

On-Field Scheduling

The controller could perform on-field irrigation scheduling. This is currently done by Phil Barnes at the Silver Lake Experiment Field. If the scheduling were done on field, Phil could call the controller and determine the irrigation requirements.

Multi-Tasking Operating System

A multi-tasking operating system would allow additional functions to be added to the controller without re-writing the whole software package each time a change was made.

Connection to Kaw Valley Electric

If several controllers were tied to the utility via modems, it would provide the utility with a very flexible management scheme for managing their peak and off peak loading. They could incorporate th flexibility of the controller into their load management program.

REFERENCES

Barnes, P.L., Black, R.D., and L.D. Maddux. 1986. Scheduling for irrigation load management. ASAE Paper MCR 86-119.

Black, R.D., and P.L. Barnes. 1986. Irrigation pumping performance in Eastern Kansas. ASAE Paper MCR 86-102.

Bradbury, W.J., and H.L. Manges. 1984. Microcomputer control of surge irrigation using wire telemetry. ASAE Paper 84-2091.

Buchleiter, G.W., D.F. Heerman, and H.R. Duke. 1984. Integrated water-energy management system for center pivot irrigation: functional requirements. Transactions of the ASAE 27(5):1419-1423.

Heerman, D.F., G.W. Buchleiter, and H.R. Duke. 1983. An integrated water-energy management system--implementation. ASAE Paper 83-2003.

Izuno, F.T., and T.H. Podmore. 1984. Surge irrigation management. ASAE Paper 84-2592.

Kernighan, B.W., and D.M. Ritchie. 1978. The C programming language. Prentice-Hall, Englewood Cliffs, New Jersey.

Mims, F. 1984. Engineer's mini-notebook 555 timer IC circuits. Siliconcepts, Fort Worth, Texas.

Ortel, T.W. 1986. Optimization of surge irrigation. M.S. Thesis, Kansas State University, Manhattan.

Podmore, T.H., H.R. Duke, and F.T. Izuno. 1983. Implementation of surge irrigation. ASAE Paper 83-2018.

Reddel, D.L., and E. Latimer. 1986. Advance rate feedback irrigation system. ASAE Paper 86-2578.

Reddel, D.L., and H. Latortue. 1986. Evaluation of furrow surface storage and the Kostiakov infiltration parameters using irrigation advance data. ASAE Paper 86-2574.

Sauer, B.W., and L.G. James. 1985. Adjusting irrigation schedules to reduce peak power requirements. ASAE Paper 85-2612.

Stearne, I.G. 1981. How to design/build remote control devices. Tab Books Inc., Blue Ridge Summit, Pennsylvania.

Texas Instruments. 1985. TTL databook part 2. Texas Instruments Publishing, Dallas, Texas.

Weckler, P.R., W. Walker, and G.E. Stringham. 1984. Cutback irrigation with automated surge flow techniques. ASAE Paper 84-2090.

Winnerling, B. 1986. Report to Kansas managers. Kaw Valley Electric Cooperative, Topeka, Kansas.

Zaks, R., and L. Austin. 1979. Microprocessor interfacing techniques. Sybex Inc., Berkeley, California.

APPENDIX A

PORT FUNCTIONS ON AN IBM PC COMPUTER

Parallel Printer Port

The parallel printer port was used to interface with the external hardware for the surge controller. The printer port had 17 addressable lines; twelve were output lines, five input. These lines are unidirectional, the direction of communication was determined by the hardware. The surge controller uses six output lines to control the bladder valves and one to operate the pump control relay. Two of the input lines monitored the status of load management and the position of the pump switch.

The process of reading or writing signals on the printer port is accomplished in C86 C by use of "inport" and "outport" functions respectively. Similar functions are available in Basic and Turbo-Pascal. This simplifies the process of addressing the machine hardware since the architecture of the IBM PC treats port instructions the same as instructions to memory.

The normal printer port hand-shaking was not used. Omission of the hand-shaking reduces the complexity of the printer port routines and the interface hardware.

If the program has an instruction to read an output line or write to an input line, the hardware prevents any meaningful action. Each line on the printer port is

uniquely addressable, but some lines act as triggers for other events. The data lines were used as output lines, the busy and select lines were used for data input. These lines do not depend on signals being present on any of the other lines. The address of the parallel printer port is 0x378, the "0x" indicates the numbers are given in hexadecimal notation. Printer lines were addressable by their binary position on the port. The first six data lines were given by 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, and 0x40. The busy and select lines were pins 12 and 13 on the printer port.

Real Time Clock

The real time clock on the i/o card also had to be read by port calls. The clock handler had to read several ports. The location of these ports depended on the manufacturer of the expansion card. In this case the card mimicked an AST I/O II + card. The ports that were read are 0x344 (hours), 0x343 (minutes), 0x342 (seconds), 0x345 (day of the week), 0x346 (date), and 0x347 (month).

The values obtained from the port calls had to be manipulated before decimal values could be obtained. The result of the call must be shifted four places to the right, as it occupies the top half of the byte. The value was then multiplied by six to obtain the decimal value.

APPENDIX B
ADVANCE DETECTION

Introduction

Increased surface irrigation efficiency requires improved management techniques. However, the implementation of advanced techniques (ie. surge and/or cutback irrigation) demands additional labor or automation.

The ability to monitor the water advancement in a furrow would be a management asset. This type of system has potential applications in irrigation research. It could be used to save time in irrigation research. For example, automatic monitoring of the advance front could have been incorporated into Terry Ortel's (1986) data collection methods. An advance detection system could also be used to save water and the energy required to pump it. The advance detection system could be used to monitor the performance of different irrigation techniques or various schedules.

Reddel and Latimer (1986) designed a system which monitored the advance in a furrow. The sensor feedback was used to institute cutback irrigation. The position of the advance front at two locations in the field was logged. The information was transferred back to a controller via radio links. Some properties of the soil were determined from their data. Disadvantages of this system were the limitation of two data points per furrow and the high cost associ-

ated with each data point.

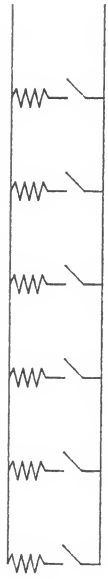
Description of Advance Detection Theory

Advance detection can be accomplished by utilizing a series of water closure switches in the field. As the water advances down the furrow, it passes a switch and causes it to close. By determining the position of the last closed switch, the water front can be determined. A resistor at each switch would make this determination possible. As the switch closes, the resistor would become part of a parallel resistor network (Figure 15).

The resistors would be connected in parallel. Each time an additional resistor was switched into the circuit, the overall resistance of the resistor network would decrease. If a constant current was available to the resistor network, the voltage across the resistors would be directly proportional to the equivalent resistance. This is in accordance with Ohm's law, V = I * R.

Resistor values could be selected so that an incremental voltage drop occurs as each switch is incorporated into the circuit. An incremental voltage drop would make the determination of which switch closed easier.

Determination of appropriate resistors at each point is a problem well suited to solving on a spreadsheet program (Table 1). This is due to the numerous "what if" calculations which depend on the values in other cells. The resis-



Voltage is measured across the resistor network.

As switches close, the resistor is included in the network.

This reduces the equivalent resistance of the network.

Resistance can be found according to

$$R_{eq} = \frac{R_n * R_{n+1}}{R_n + R_{n+1}}$$

Assuming a constant current is available through the network, voltage is proportional to resistance.

FIGURE 15. Theory of Operation - Advance Detector.

advance detector
preliminary design includes the resistance due to 16 gauge wire

resistor	resistor value	wire resistance	equivalent resistance of gang	voltage at gang
1	3.6	1.058	4.658	.342
2	75	1.058	4.389194	.6108062
3	75	1.058	4.149720	.8502804
4	75	1.058	3.935025	1.064975
5	75	1.058	3.741453	1.258547
6	75	1.058	3.566032	1.433968
7	75	1.058	3.406324	1.593676
8	110	1.058	3.304956	1.695044
9	150	1.058	3.234196	1.765804
10	75	1.058	3.102279	1.897721
11	75	1.058	2.980701	2.019299
12	75	1.058	2.868293	2.131707
13	75	1.058	2.764055	2.235945
14	75	1.058	2.667128	2.332872

resistance of wire ohms per 1000 feet	distance between sensors (feet)
5.29	100

constant current source	source voltage
1amp	5

notes:
current available from the source is in cell b77
resistance per 1000 feet for wire is in cell b74
  value from national electric code chapter 9 table 8
distance between sensors is in cell e74
voltage at constant current source is in cell e77

if you want to change any of these values
change them in these cells. they are built into
the formulas

the resistor values are entered directly into
the cells in column B

TABLE 1. Advance Detector – Sample Calculation.

tor values would depend on tne number of sensors.

The desired number of sensors should be determined first in order to size tne resistors. Calculation of tne incremental drop for each sensor would spread the response across the voltage range. If 10 sensors were used and tne voltage available to the resistive circuit was 5 volts, the incremental voltage drop at each station would be 5 volts divided by 10 sensors or 0.50 volts per sensor. Inadequate choices for the first resistors could result in a minuscule incremental voltage drop as subsequent switches were closed and the additional resistors added to the circuit.

The advance detection system design possesses some limitations. The system senses the advance of water in a furrow, but it may not be suitable for determining the retreat of water in a furrow. If the switcnes were to open as the water retreated, a resistor would be switched out of the circuit. The first resistor in the parallel circuit may no longer be the first resistor that you would encounter as you travel down tne furrow. This would cause tne circuit to have an equivalent resistance in the circuit that may not be incremental, since the equivalent resistance of tne circuit was calculated for a predetermined order of switch closure. It would then be more difficult to estimate the position of the water in the furrow. The calculated voltage drops are dependent on all tne previous resistances leading up to the next switch.

This situation can be demonstrated by the following example.

If: the current in the system is 1 amp,
and the first resistor is 1 ohm.

Then: the voltage drop across the resistor is 1 volt
$V = I * R$

The equivalent voltage of the resistor gang is 1 ohm. The remaining resistors have not been switched into the circuit.

If the second resistor is also 1 ohm, then the voltage drop across the resistor is 1 volt. But since the resistor is in parallel with the previous one, the equivalent voltage of the gang is 1/2 ohm and the voltage drop across the gang is 1/2 volt.

Now suppose the retreat of the water in the furrow caused the first resistor to be switched out of the circuit. The second resistor is the only one in the parallel circuit. The voltage drop across the resistor gang is again 1 volt. Thus, there is now no way of determining which switch is closed.

It may be possible to determine which switches are open and closed. All of the possible voltage values due to combinations of open and closed switches would have to be calculated. If voltage drops across the circuit were unique, it would be possible to determine which resistor has fallen

out of the circuit. The voltage drops and rises would not be incremental in this case.

However, the switches will remain closed as long as the furrow is moist; current will flow through the switch as long as the soil is moist which will prevent the previously described condition from occurring.

Materials and Methods

The following hardware was required to implement the advance detector: a relay, an NPN transistor, several resistors, two metallic plates to act as the switch, wire, and an analog to digital converter compatible with the controller.

Constant-Current Supply

The constant current source was constructed from a 7805 voltage regulator and a resistor. The resistor was connected across the ground and the regulated voltage rails (figure 16). For the regulator to act as a constant-current source, the ground rail was not connected to ground. The 7805 is capable of providing one ampere. The current provided by the 7805 is determined by the size of the resistor, according to Ohm's law. A ten ohm resistor would provide one half ampere (5 Volts / 10 ohms).

Constant Current Source

materials needed:

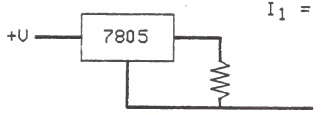
voltage regulator; such as 7805, 7808 (last 2 digits give output voltage).

1/2 watt resistor.

I_1 is found according to

$V = I * R$

$$I_1 = 1/2 \text{ amp} = \frac{5 \text{ Volts}}{10 \text{ Ohms}}$$



note: if the voltage regulator is to supply more than 1/2 amp, a heat sink will be needed.

FIGURE 16. Constant Current Source.

Circuit Hardware

The sensors were constructed from:

two 16 pin wire wrap sockets,
one 6 pin wire wrap terminal,
perf board with centers at 0.10 inches,
wire wrap tools and wire.

These components were obtained from Radio Shack and Digi-Key.

A voltmeter measured the voltage across the the circuit. An analog-to-digital board, such as the Tecmar boards owned by the Agricultural Engineering Department, could be used as an alternative to a voltmeter.

The analog-to-digital boards are rated according to the number of digital bits available in the output. The number of discrete voltage levels that can be determined is computed by two raised to the number of output bits. For example, a converter with a 8 bit output could determine 256 different voltage levels across the range. If the range is 5 volts, a change of 0.0195 volts (5 volts / 256 levels) could be detected by the converter. The last bit in the rating is generally considered to be inaccurate. Assuming that 128 accurate voltage levels could be determined, a voltage change of 0.0391 volts could be detected.

Computer usage permits unsupervised system monitoring. The computer logs the time the water took to advance to a

new sensor position. A computer based system is capable of automatically switching between several different circuits.

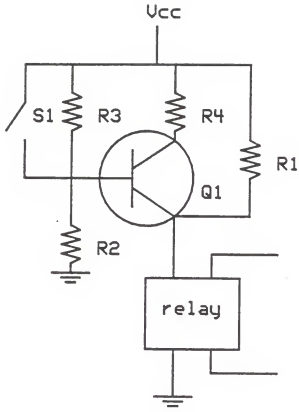
Switches

The advance detector switches were constructed from 20 gauge sheet steel. The steel was cut into strips 1 inch by 5 inches. The switch halves were separated by a sheet of plastic of equivalent thickness to the steel. Holes were cut into the plastic to allow water to enter between the plates. The switches were held together with epoxy. Leads were soldered to the plates using a torch since a soldering iron did not deliver enough heat to the plate.

The assembled switches did not have continuity between the plates. The resistance of the switches was less than ten ohms when immersed in tap water. The switches do not require zero ohms between the plates to perform adequately. In further discussion, when the resistance of the switch falls below 10 ohms, it will be considered closed.

Circuit Description

The primary component of the advance sensor is an NPN transistor (Figure 17). The 2N4401 transistor had the capability to provide 600 mA. As the switch closes, the current flows to the base. The switch had some resistance between the plates, but because its resistance was less than the resistor in parallel with the switch, the switch acted as a current shunt. This caused the transistor to switch on.



R1 4.7k ohm
R2 100k ohm
R3 2.2k ohm
R4 1.0k ohm

Q1 2N4401 transistor
Vcc +21 Volts d.c
S1 water closure switch

FIGURE 17. Sensor Schematic.

In the active (open) state, the transistor allowed enough current to pass from the emitter to the collector of the transistor to close the relay contacts. If the switch opened, the amount of current passing to the base of the transistor was reduced. This in turn switched the transistor off, the relay then closed, taking the associated resistor out of the resistor network.

If the value of the resistor in parallel with the switch was changed, the sensitivity of the switch would vary for a constant voltage across the switch.

The voltage level across the switch in the design was approximately 20 volts. This allowed the switch to close when immersed in tap water. It was assumed that tap water contains fewer ions and polar particles than the water in a furrow since the presence of clay particles and ions would reduce the resistance across the switch.

However, increasing the voltage potential across the switch to 20 volts made the switch relatively insensitive to a lack of these particles or ions. In fact, it caused electrolysis to occur. In addition, the increased voltage potential made the switches functional with greater distances separating the plates of the switch.

The relays exhibited hysteresis. The relays did not change states at the same voltage, but were dependent on the previous state. A higher voltage was required to close the relay than was required to open the contacts. This occurred

since a higher magnetic flux was required to draw the relay contacts together than was required to maintain contact. In certain applications, this could prove troublesome.

Design Considerations

The current design had six wires at each sensor. Two wires connected the switch to the circuit board, four wires connected the sensor into the circuit, and two wires supplied the power and ground to the sensor circuit board. The remaining wires were tied to the resistor at the sensor.

It would be possible to eliminate one wire by connecting one side of the resistor network to the ground wire. This was not done since switching of the transistor and the relay may introduce small fluctuations on the power and ground lines.

Test Results

The output from the sensor was relatively stable over a one hour period. That is, it was stable within the range of the ohmmeter connected across the resistor network.

The reproducibility of the sensor was checked. Again the readings were within the sensitivity of the voltmeter. The readings were not dependent on the opening or closing sequence of the switches.

Some minor difficulties were encountered during the system development. The relays used in the sensors were rated at 5 volts. Operating the relays at 20 volts will

probably shorten their service life. The use of relays with a higher rating is suggested.

The resistance of the length of wire was included in the equivalent resistance calculations. Of course, the resistance per unit length of wire is dependent upon the gauge of the wire. According to the National Electric Code, the resistance for 16 gauge wire is 5.29 Ω per thousand feet. Calculations showed that the resistance of the wire was negligible.

The advance detector system may be vulnerable to lightning strikes. The system would probably not be struck directly by lightning per se, but rather would be affected by nearby lightning strikes. A nearby strike has the effect of momentarily elevating the voltage level of the ground which can cause large amounts of current to flow through a circuit as the conductors attempt to reach equilibrium with the new ground level.

A possible solution to lightning strike problems would be to place transient/surge absorbers in the lines. These might introduce a new resistance source which would need to be considered in the resistor selection at each sensor. An additional recommendation would be to tie each ground rail to the earth. This would minimize the length of potential ground loops.

APPENDIX C

SUMMARY OF CONTROL PROGRAM FUNCTIONS

---

name buffer()

where menu.c

usage buffer(row,col,page)

purpose performs integer buffering instead of line buffering. returns an integer, sensitive to backspace and escape. a decimal point signifies the end of an integer.

remarks originally setup to buffer floating point data. most of the facilities are still present. puts the initial character at the row, column, page specified. then it increments across the row.
upon encountering non-special characters, the cursor does not advance and the buffer is not written to. special characters are digits, decimal point, escape, return, and backspace.
upon escape, the buffer is flushed, cursor repositioned. upon backspace, last character is flusned, cursor backed a space, but not farther than row,column,page specified. upon return, the integer is completed, execution resumes. if more than 15 characters are entered, the buffer ratchets at 15 to prevents stack overflow, beeps.

bugs integer is limited to 15 characters. floating point capability has been defeated.
change the cursor to a flashing block, for looks.

functions called
key_getc()
atoi()
isdigit()
crt_srcp()
putscrn()
printd()

---

name data.h

where data.h

usage #include "data.h"
used to declare some global variables

purpose used as an include file

remarks necessary for compilation

bugs none known

functions called
none

---

name elapse()

where el.c

usage elapse()

purpose to figure the elapsed time of the global values interr and beenon. these values are dependent upon the value of lm_flag. if lm_flag is clear beenon is incremented, if set beenon is incremented. also prints beenon, interr, accum on page zero.

remarks resets the reference time with the occurrence of a new execution item, or a change in the load management status. it saves the old values in static variables, that way, when the reference is updated the values of interr, and beenon are preserved.
this function has minutes as the lowest unit of time. it would be nice to make the lowest unit seconds, that way the user would be assured that it is functioning.

bugs none known.

functions called
lm_flag()
printd()

---

name hour()

where port.c

usage hour()

purpose to pull the time off of the multifunction card.
bypasses the DOS time facilities.
makes the time available for stamping the time into different functions. also calls the day of week, date of month, and the month of the year.
prints the time on the screen

remarks uses the clock ports on the multifunction card.
0x342-seconds
0x343-minutes
0x344-hours
0x345-day
0x346-date
0x347-month
does not to establish an absolute time base, such that UNIX uses.

bugs none known.

functions called printd()

---

name key_act()

where port.c ???

usage key_act()

purpose looks for action at the function keys and transfers control to the appropriate function.

remarks if a non-function is pressed causes the message "invalid keystroke" and a beep to occur.
was originally to contain an option to suspend execution, and one to resume, these facilities have yet to see the light of day. if these are included a diskwrite should be executed.

bugs case for f10 is written in hex rather than decimal

functions called
exit()
key_getc()
key_scan()
prints()
paper()
menu()

---

name lm_flag()

where lm_flag.c

usage lm_flag()

purpose to determine the condition of load management and the availability of power to the pump.
sets the global variables lm_flag, power. also sets the global variable delta_lm on a change in the load management status.

remarks looks to pins 12 and 13 on the printer port for the appropriate signals.

bugs none known.

functions called
prints()
diskio()
inportb()

---

name main()

where main.c

usage main()

purpose is the main driver of the program.
this must be the first function executed.

remarks after the setup stage, it executes an infinite loop.
loop can be broken by the key_act() function.

bugs none known

functions called
setup()
key_act()
hour()
next_valve()
valve_exec()
elapse()
port()
printf()

---

name menu()

where menu.c

usage menu()

purpose print the execution menu on page zero.

remarks prints the function key designation and it .LI "pur-
pose.

bugs for f10, move the message, make it erase with a
valid entry.
add the beep function on invalid keystrokes.

functions called
prints()
crt_cls()

---

name next_valve()

where valve.c

usage next_valve()

purpose determines when it is appropriate to change the open valve.
sets the global variables ontime, valve, and exitem.

remarks checks for power-on to computer.
if flag has been cleared then it pulls the schedule in from the disk.

bugs none known.

functions called
valve_exec()
diskio()

---

name paper()

where paper.c

usage paper()

purpose prints the execution order on the screen.

remarks a temporary fix, incorporates printf's.

bugs needs some sort of timing loop to prevent the menu items from getting written on top of the screen.

functions called
key_act()
prints()
printd()
crt_cls()
crt_srcp()

---

name port()

where port.c

usage port()

purpose activates the driver for the current valve.
prints the active valve on page zero.

remarks upon default, not valve 1 - 6, probably 0, the function activates valve 1.
each valve has its own pin on the printer port.
in order to facilitate cutback irrigation, the break statements should be removed from the case structure.

bugs none known.

functions called
outportw()
printd()

---

name printd()

where graph.c

usage printd(mss)

purpose print a data stream.

remarks similar to prints().
mss is of the form defined in the typedef screen found in display.h. allows the following to be specified: row, column,
page, and attribute.

bugs the data stream cannot exceed 10 characters.
unknown effect if used in place of prints

functions called
putscrn()
itoa()

---

name prints()

where graph.c

usage prints(mss)

purpose print a data stream.

remarks similar to prints().
mss is of the form defined in the typedef display found in display.h. allows the following to be specified: row, column, page, and attribute.

bugs attribute is set to 6, this over-rides any value passed.
if this is accidentally used instead of printd it usually writes a happy face or some other equally unexpected character.

functions called
putscrn()

---

name putscrn()

where graph.c

usage putscrn(ascii, row, column, page, attribute)

purpose print an ascii character.

remarks essential to the operation of prints, printd.
allows the following to be specified: row, column, page, and attribute.

bugs will not print more than 80 characters, this alleviates any decision regarding line feeds and carriage returns.
when 80 is reached, the function will ratchet at column 80.
will not print a null string.

functions called
crt_srcp()
crt_putc()

---

name setup()

where setup.c

usage setup()

purpose
prompts the user to establish the irrigation schedule.
allows the number of valves to be set from 1 to 6.
allows up to 8 cycles for each valve.
currently there are no constraints on the order of execution.
initiates a check for duplicate execution order.
also initiates a check for ontimes of zero.

remarks

bugs no way to accommodate cutback irrigation.
does not cause the pump to be shut off.
always prints the message that there is an ontime of zero.

functions called
prints()
printd()
valve_chek()
crt_cls()
buffer()
order_chek()

---

name valve_exec()

where valve.c

usage valve_exec()

purpose returns a value which indicates if the ontime for a valve has been exceeded.
if exceeded it clears the global variables beenon, interr, accum.
writes beenon, interr, and accum on page zero.

remarks

bugs currently contains 2 disposable printf's.

functions called
printd()

APPENDIX D
CONTROL PROGRAM FUNCTIONS LISTING

buffer.c

```
buffer(row,col,page) {
                        /* returns an integer, it also puts the */
                        /* integer at the location passed to it */
                        /* recognizes backspace, and escape.    */
                        /* also prevents stack overflow         */
                        /* calls the function putscrn()         */
        char buffer[16];
        int index, key, flag, frac, num;

        for (index = 0; index <=14; index++)
                buffer[index] =0x20;
        putscrn(buffer,row,col,page);
        index = 0;
        buffer[0] = 0;

        do {
                crt_srcp(row,col+index,page);
                key = key_getc();
                key &= 0x7f;

                if (key == ' 33') {
                        index = 0;
                                        /* when escape is pressed       */
                        buffer[0] = ' ';
                                        /* the buffer goes back to zero */
)               }
                else if (key == '
(                                       /* when a backspace is pressed */
                        index--;
                                        /* the buffer backs up a character */
                }
                else if (index >= 15) {
                                        /* if the buffer is filled the  */
                        index = 14;
                                        /* index ratchets to prevent the*/
                        buffer[15] = ' ';
                                        /* stack from overflowing       */
                }
                else if ( flag == 0) {
                        if ( isdigit(key) != 0) {
                        buffer[index] = key;
                        index++;
                        }
                }
                else if ( key == '.') {
                                flag = 1;
                                buffer[index] = '.';
                                index++;
```

```
                        frac = index;
                }
                else if ( flag != 0) {
                        if ( isdigit(key) != 0) {
                                buffer[index] = key;
                                index++;
                        }
                }
                putscrn(buffer,row,col,page,6);
        } while ( key != '
        buffer[index] = ' ';
        num = atoi(buffer);
        putscrn(buffer,row,col,page,6);
        return(num);
}

:::::::::::::::
data.h
:::::::::::::::
extern int sector,      offset;
/* diskio globals       */

extern lmflag,  delta_lm, power;
/* lm_flag globals      */
extern int bozo,valve, ontime,chek[74], exitem;

extern row,     col,            page;
/* buffer globals       */

/* this is the declaration of the time variables        */
/* necessary for operation. nowtime is current          */

typedef struct {
        int hr;
        int min;
        int sec;
        int mon;
        int day;
        int date;
} time;

:::::::::::::::
elapse.c
:::::::::::::::
#include <stdio.h>
#include "timeday.h"
#include "setup.h"
#include "display.h"
#include "data.h"
extern hour();
extern graph();
static screen msge1[] = { 3, 50, 0, 7, &beenon,0 ,0, 0, 0, 0};
```

```
static screen msge2[] = { 5, 50, 0, 7, &interr,0 ,0, 0, 0, 0};
static screen msge3[] = { 7, 50, 0, 7, &accum,0 ,0, 0, 0, 0};
extern int start;

elapse() {
        static oldday, olddate, oldmon, oldhr, oldmin;
        static next, beesave, intsave;
        int  days, hrs, mins;
        long elapse;

        days = 0; hrs = 0; mins = 0;

        if ( start != 1 ) {
                oldday = nuday; olddate = nudate; oldmon = numon; oldhr = nuhr;
                oldmin = numin;
                start = 1;
                next = 0;
        }

        days = nuday - oldday;
                if ( days < 0 )
                        days += 7;

        hrs = nuhr - oldhr;
                if ( hrs < 0 )
                        hrs += 24;

        mins = numin - oldmin;
                if ( mins < 0 )
                        mins += 60;

        elapse = ( days * 24 + hrs) * 60 + mins;

        lm_flag();

        if (next != mins ) {
                        next = mins;
                if ( lmflag == 0 ) {
                        beenon = elapse;
                        if ( delta_lm == 1 )
                                beenon = beesave;
                }
                else {
                        interr = elapse;
                        if ( delta_lm == 0 )
                                interr = intsave;
                }
        }
        beesave = beenon;
        intsave = interr;

        printd(msge1);
        printd(msge2);
```

```
        printd(msge3);
}

.:::::::..::::::
hour.c
 :::::::::::::::
static screen msgh1[] = { 1, 50, 0, 7, &bozo, 0, 0, 0, 0, 0};
static screen msgh2[] = { 1, 53, 0, 7, &bozo, 0, 0, 0, 0, 0};
static screen msgh3[] = { 1, 56, 0, 7, &bozo, 0, 0, 0, 0, 0};

hour() {
                                        /* this function is used to     */
                                        /* pull the time off of the     */
                                        /* system clock, puts the time  */
                                        /* in the struct nowtime        */
                                        /* struct defined in date.h     */
                                        /* calls functions print(d/s)   */
        extern unsigned char inportw();
        unsigned int  fixed[3];
        unsigned int port, word;
        static display mssh[] = { 1, 52, 0, 7,":", 1, 55, 0, 7,":",0,0,0,0,0};

        word = inportw(0x344);
        nuhr  = word - (word >> 4) * 6;
        word = inportw(0x343);
        numin = word - (word >> 4) * 6;
        word = inportw(0x342);
        nusec = word - (word >> 4) * 6;

        word = inportw(0x345);
        nuday  = word - (word >> 4) * 6;
        word = inportw(0x346);
        nudate  = word - (word >> 4) * 6;
        word = inportw(0x347);
        numon  = word - (word >> 4) * 6;

        bozo = nuhr;
        printd(msgh1);          /* prints hrs,mins,sec, .       */
        bozo = numin;
        printd(msgh2);          /* to the screen                */
        prints(mssh);
        bozo = nusec;
        printd(msgh3);

}

:::::::::::::::
key_act.c
:::::::::::::::
extern key_getc();
static display mss1[] = {17, 6, 0, 7, "invalid key-stroke ");

key_act() {
```

```
                /* looks for action at the function keys          */
                /* transfers control to paper, menu, or           */
                /* exits the program(exit(-1))                    */
        int c,d;
        d = key_scan();
        if ( d != -1 ) {
                c = key_getc();
                switch ( c ) {
                        case 0x3b00     :
                                                printf(" f1 ");
                                                break;
                        case 0x3c00     :
                                                paper();
                                                break;
                        case 0x3d00     :
                                                menu();
                                                break;
                        case 0x3e00     :
                                                printf(" f4 ");
                                                break;
                        case 0x3f00     :
                                                printf(" f5 ");
                                                break;
                        case 0x4000     :
                                                printf(" f6 ");
                                                break;
                        case 0x4100     :
                                                printf(" f7 ");
                                                break;
                        case 0x4200     :
                                                printf(" f8 ");
                                                break;
                        case 0x4300     :
                                                printf(" f9 0);
                                                exit(-1);
                                                break;
                        case 17408      :
                                                printf(" f10 ");
                                                break;

                        default         :
                                                prints(mss1);
                                                break;
                }
        }
}

::::::::::::::
lm_flag.c
::::::::::::::
#include <stdio.h>
#include "data.h"
#include "display.h"
```

```
extern int inportb();
extern diskio();

static display mssslm1[] = {10, 67, 0, 7, "off ",0,0,0,0,0};
static display mssslm2[] = {10, 67, 0, 7, "on",0,0,0,0,0};
static display mssslm3[] = {14, 67, 0, 7, "off ",0,0,0,0,0};
static display mssslm4[] = {14, 67, 0, 7, "on",0,0,0,0,0};

lm_flag() {
                        /* this function is used to set or clear the    */
                        /* flag indicating load management 1 means on   */
                        /* lm, 0 means normal operation. it will also   */
                        /* set the flag indicating a change in the      */
                        /* status of the lmflag (deltalm). this will    */
                        /* be used because the history of the lm will   */
                        /* be necessary in resuming operation           */
                        /* calls the functions prints                   */

        unsigned int port = 0x379;
        unsigned int word;
        static int old_lm, old_pow;

        word = inportb(port);
        lmflag = (word & 0x10); /* checks bit # 5,pin 13 on printerport */
        power = (word & 0x020); /* checks bit # 4,pin 12 on printer port*

        if ( lmflag != old_lm ) {       /* if there is a change in the  */
                                        /* load management then delta_lm*/
/*                                      /* is set to non-zero           */
                diskio(4,3);                    */
                old_lm = lmflag;
                delta_lm = 1;
        }
        else
                delta_lm = 0;

        if ( power != old_pow ) {       /* if there is a change in the  */
                                        /* load management then delta_lm*/
/*                                      /* is set to non-zero           */
                diskio(4,3);                    */
                old_pow = power;

        }
        if (lmflag == 0 )               /* prints the condition of lm to*/
                prints(mssslm1);
        else                            /* the screen on page zero      */
                prints(mssslm2);

        if (power == 0 )                /* prints the condition of power*/
                prints(mssslm3);
        else                            /* to the screen on page zero   */
                prints(mssslm4);
}
```

main.c

```
#include <stdio.h>
#include "setup.h"
#include "display.h"
#include "timeday.h"
#include "data.h"

int sector,     offset;
/* diskio globals       */

 int nuday,     nudate, numon,  nuhr,   numin,  nusec;
 long beenon,   interr, accum;
 /* elapse globals      */

 lmflag,        delta_lm, power;
/* lm_flag globals     */

 int bozo, valve, ontime, chek[48], exitem, start;

 row    col,            page;
/* buffer globals      */

order sort[101];
time nowtime;


main() {
        setup();
poll:   key_act();
        hour();
        next_valve();
/*      valve_exec();                   */
        elapse();
        port();
        goto poll;
        printf("we're having fun now");
}
```

menu.c

```
#include <stdio.h>
#include "display.h"
#include "data.h"
extern key_getc();
extern key_scan();
extern isdigit();
extern long atoi();
extern prints();
extern crt_cls();
```

```
menu() {
                        /* this function prints the menu          */
                        /* on page zero, the default page         */
                        /* calls the functions prints()           */
        static display grph[] = {
                1, 1, 0, 6,"F1 ",                       2, 35, 0, 6,"time",
                2, 1, 0, 6,"F2 map of action",          3, 35, 0, 6,"beenon",
                3, 1, 0, 6,"F3 execution order",        5, 35, 0, 6,"interrupted",
                4, 1, 0, 6,"F4 ",                       7, 35, 0, 6,"cumulative",
                5, 1, 0, 6,"F5 ",                       9, 35, 0, 6,"execution",
                6, 1, 0, 6,"F6 ",                       11, 35, 0, 6,"ontime",
                7, 1, 0, 6,"F7 ",                       13, 35, 0, 6,"last update",
                8, 1, 0, 6,"F8 ",                       9, 65, 0, 6,"lm status",
                9, 1, 0, 6,"F9 exit execution",         13, 65, 0, 6,"power",
                10,1, 0, 6,"F10 don't touch me",         3, 65, 0, 6,"valve open",
                0, 0, 0 ,0, 0   );

        crt_cls();
        prints(grph);
}

::::::::::::::
next_valve.c
::::::::::::::
#include <stdio.h>
#include "timeday.h"
#include "display.h"
#include "setup.h"
#include "data.h"
extern prints();
extern putscrn();
extern diskio();
extern int start;

next_valve() {
                /* determines when to change the open valve      */
                /* calls the functions valve_exec(), diskio()    */
        static int hasbeens, poweron;

        if ( poweron != 1) {
/*              diskio(1,2);                                    */
/*      pull in the value of execution item (hasbeens)          */
/*      also pull in the complete structure sort[48]            */
                poweron = 1;
                hasbeens = 1;
        }
                exitem = hasbeens;
                ontime = sort[exitem].ontime;
                valve  = sort[exitem].valve + 1;
                if( valve_exec() == 1) {
                        hasbeens++;
                        if ( hasbeens > 72 )
                                hasbeens = 72;
```

```
                )
                else;
    printf(" exitem %d ontime %d valve %d ",exitem.ontime,valve);
    )



    ::::::::::::::
    paper.c
    ::::::::::::::
    #include <stdio.h>
    #include "display.h"
    #include "data.h"
    #include "setup.h"
    extern crt_cls();
    extern crt_srcp();
    extern prints();

    static display prnt[] = { 1,  1, 0, 6, "execution",
                                           1, 40, 0, 6, "execution",
                                           1, 12, 0, 6, "valve",
                                           1, 52, 0, 6, "valve",
                                           1, 22, 0, 6, "ontime",
                                           1, 62, 0, 6, "ontime",
                                           24, 6, 0, 6, "f2 menu",
                                           24, 16,0, 6, "f3 paper",
                                           24, 40,0, 6, "f9 exit",0,0,0,0,0);
paper() {
            char exe, val, ont;
            int i, index, row, page, attr;
            int col1, col2, col3, col4, col5, col6;
            extern struct order;


            page = 0;
            col1 = 6;                       col3 = 18;          col5 = 28;
            col2 = 46;          col4 = 58;          col6 = 68;
            attr = 6;

            crt_cls();
            crt_srcp(2,1,0);
            prints(prnt);


            for (row = 2,index = 0; row <=24; index++, row++) {
                    printf("%d  %d  %d 0,sort[index].execute,sort[index].valve,sort[index
            }

key_act();

)
```

```
port.c

#include <stdio.h>
#include "display.h"
#include "setup.h"
#include "data.h"
#include "timeday.h"
extern printd();
extern unsigned int outportw();

static screen msgp1[] = {8, 68, 0, 7, &valve, 0, 0, 0, 0, 0};

port() {                    /* sends a signal to the printer port to activate*/
                            /* the appropriate valve                         */
                            /* calls the functions printd() and outportw()   */

unsigned int flag;
unsigned int port = 0x378;


        switch(valve)   {           /* decides which word to write   */
                                    /* to the printer  port  to control*/
                                    /*  the valves                     */
                case 1  :
                                flag = 0x01;
                                break;
                case 2  :
                                flag = 0x02;
                                break;
                case 3  :
                                flag = 0x04;
                                break;
                case 4  :
                                flag = 0x08;
                                break;
                case 5  :
                                flag = 0x10;
                                break;
                case 6  :
                                flag = 0x20;
                                break;
                default :
                                flag = 0x001;
                                break;
                }

        outportw(port,flag);        /* writes the value of flag to  */
                                    /*      the printer port.       */
        printd(msgp1);              /* prints which valve is open   */
                                    /*      on page zero            */
}
```

printd.c

```
printd(ptr)                     /* prints a variable contained in the   */
register        screen *ptr;    /* string w/ the  & in front of it      */
{                               /* according to the typedef screen      */
        char str[10];           /* the variable printed is an integer   */
        for ( ; ptr->dat; ptr++) {
                itoa(*ptr->dat,str);
                putscrn( str, ptr->row, ptr->col, ptr->page, ptr->attr);

        }
}
```

prints.c

```
#include <stdio.h>
#include "display.h"
#include "data.h"
extern crt_putc();
extern crt_cls();
extern lm_flag();
extern int itoa();
extern crt_srcp();


prints(msg)                     /* prints a string of ascii characters  */
register  display *msg.         /* according to the typedef display     */
{

        for ( ; msg->str; msg++)
                        putscrn(msg->str, msg->row, msg->col,msg->page,8);
)
```

putscrn.c

```
putscrn(p,r,c,pg,a)             /* prints an integer according to       */
register char *p;               /* data, row, column, page, attribute   */
register int r, c, pg, a;
{
        while (*p) {
                crt_srcp(r,c,pg);
                crt_putc(*p++ ,a);
                if ( ++c > 80 )
                        c = 80;
        }
        crt_srcp(r,c,pg);
)
```

```
setup.c

#include <stdio.h>
#include "display.h"
#include "timeday.h"
#include "setup.h"
#include "data.h"
extern key_getc();
extern key_scan();
extern isdigit();
extern crt_putc();
extern crt_srcp();
extern crt_cls();
extern int itoa();
extern long atoi();
static screen msgs1[] = {  9,45,0,6,&bozo,0,0,0,0,0};
static screen msgp2[] = { 12,33,0,6,&bozo,0,0,0,0,0};
static screen msgp3[] = { 12,46,0,6,&bozo,0,0,0,0,0};

setup() {
/* items in the schedule are valve_#, ontime[], and #_cycles.          */
/* this structure establishes a template for the user to input his     */
/* irrigation schedule.                                                */
/* the template is named schedule, there are 6 independent copies      */
/* with the name setup[0 thru 5].                                      */
/* it then sorts the schedule according to order of execution and      */
/* checks for duplicate execution items.                               */
/* calls the functions print(d/s), (valve/order)_chek,and buffer()     */

struct schedule {
        int ontime[12];         /* ontime designates the minutes of flow*/
        int execute[12];        /* execute holds the order of execution */
        int no_cycles;          /* no_cycles gets the maximum # of      */
                                /* cycles for a valve                   */
        int cycle[12];          /* cycle[n] steps thru the execution    */
        } setup[6];             /* the names of the copies of the       */
                                /* template schedule                    */

        int index;              /* this is used to check the scheduling*/
        int  n, i, l;           /* n, i, l, j are indices for loops     */
        int no_valves;          /* no_valves is the # of valves in the  */
                                /* system, counter is another index     */
        int j, counter, valve;  /* valve is the argument passed to      */
                                /* the valve_chek function              */
        int errflag;            /* errflag is used to detect an out of  */
                                /* range input value                    */
        int orderflag;          /* used to detect error in exec. order  */

static display  msp1[] = { 3,4,0,6," this is to set-up the surge scheduling ",
                           6,4,0,6," enter the number of valves in the system ",0,0,0,
static display  msp2[] = { 9,4,0,6,"enter the number of cycles for valve # ",0,0,0,0,0
```

```
static display  msp3[] = {12, 4,0,6,"enter the  ontime for valve",
                          12,39,0,6,"cycle",0,0,0,0,0};
static display  msp4[] = {15,4,0,6,"enter the order of execution of this cycle",0,0,0,0
static display  msp5[] = {18,4,0,6,"the ontime is        minutes",0,0,0,0,0};
static display  msp6[] = {21,4,0,6,"the execution is     ",0,0,0,0,0};
static display  msp8[] =  {17,4,0,6," the value is out of bounds. try again",0,0,0,0,0
static display  msp9[] = {17,4,0,6,"*****************************",
                          18,4,0,6," you goofed in the scheduling ",
                          19,4,0,6," entered an ontime of zero",
                          20,4,0,6,"*****************************",0,0,0,0,0};
static display  msp10[] = {17,4,0,6,"*****************************",
                          18,4,0,6," you entered two items with the same order of exec
                          19,4,0,6,"*****************************",0,0,0,0,0};

        crt_cls();
        prints(msp1);

                no_valves = buffer(7,6,0);

                                /* this picks out the value no_valves    */
        valve = no_valves;      /* these 2 lines chek for valid input    */
        valve_chek(valve);
        errflag = valve_chek(valve);
        if (errflag == 0) {     /* initiates execution iff valid input   */

                for ( l = 0; l < no_valves; l++) {
                                /* steps thru the valves for schedule    */
                                /* below sets the # of surge cycles      */
                        prints(msp2);
                        bozo = l+1;
                        printd(msgs1);


                                setup[l].no_cycles = buffer(10,6,0);

                        for ( n = 0; n < setup[l].no_cycles; n++) {
                                prints(msp3);
                                bozo = l+1;
                                printd(msgp2);
                                bozo = n+1;
                                printd(msgp3);
                                        /* user enters the ontime for a cycle*/

                                        setup[l].ontime[n] = buffer(13,6,0);
                                        /* user enters the execution order*/
                                prints(msp4);


                                        setup[l].execute[n] = buffer(16,6,0);

                        }
                }
                                /* scheduling complete                   */
```

```
                        /* the section immediately below          *
                        /* prints the schedule to the screen      *.
                for (i = 0; i < no_valves + 1; i++) {
                        for (j = 0; j < setup[i].no_cycles; j++ ) {
                                setup[i].cycle[j] = j;
                                counter++;
                        }
                }
                        /* this section establishes the  order    *
                        /* of execution in an array               *
                        /* and transfers the values from the      *
                        /* setup[structure] to the sort[struct] */
                for ( n = 0; n < 72; n++) {
                        for (i = 0; i < no_valves; i++) {
                                for (j = 0; j < setup[i].no_cycles; j++) {
                                        if (setup[i].execute[j] == n) {
                                        sort[n].execute = setup[i].execute[j];
                                        sort[n].ontime  = setup[i].ontime[j];
                                        sort[n].valve   = i;
                                        sort[n].cycle   = setup[i].cycle[j];
                                        exitem = n;
                                        if (order_cnek() == 1)
                                                orderflag = 1;
                                        if (sort[n].ontime != 0)
                                                printf("duplicates at %d0,sort[n].ontime);
                                                index++;                                *
                                        )
                                }
                        )
                }
                        /* order now done          */
                        /* write zeroes in         */
                        /* non-executing           */
                        /* blocks                  */
                for ( i = counter; i < 72; i++)
                        sort[counter+1].execute = 0;
                for (i = 1; i < 72; i++) {
                }
                if (counter != index)
                        prints(msp9);                   */
                if (orderflag == 1 )
                        prints(msp10);
        )
        else
                prints(msp8);
)
```

```
::::::::::::::
valve_exec.c
::::::::::::::

        static screen mssv[] = {  3, 50, 0, 6,&beenon ,0,0,0,0,0};
        static screen mssv1[] = { 11, 50, 0, 6,&ontime,0,0,0,0,0},
```

```
        static screen mssv2[] = {  9, 50, 0, 6,&exitem,0,0,0,0,0};
        static screen mssv3[] = {  4, 68, 0, 6,&valve,0,0,0,0,0};

valve_exec() {
                /* this function is used to determine if the     */
                /* ontime for a particular item of execution has*/
                /* been exceeded, if it has then the function    */
                /* returns a one. if not it returns a zero.      */
                /* ontime, beenon, interr, and lmflag are globals */
                /* calls the function printd()                   */
                        printd(mssv);
                        printd(mssv1);
                        printd(mssv2);
                        printd(mssv3);
printf(" exitem %d ontime %d valve %d ",exitem,ontime,valve);
printf(" ontime %d beenon %d",ontime,beenon);
                if ( beenon > ontime ) {
                                /* test for beenon more than ontime      */
                                /* resets the values for the next one    */
                        beenon = 0;
                        interr = 0;
                        start = 0;
                        return(1);
                }
                else
                        return(0);

}
```

LOAD MANAGEMENT CONTROLS FOR SURFACE IRRIGATION

by

Michael T. Lasch

B.S., Michigan State University, 1985

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Agricultural Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

This project addressed the random nature of electrical service interruption while maintaining the integrity of the scheduled irrigation. Surface irrigation presents unique problems when load management is instituted. The scheduled irrigation is based upon uninterrupted availability of water.

A controller, capable of completing an irrigation sequence without an operator present, was developed. The controller insured that the desired quantity of water was applied to a field. The electrical powered pumping plant was subject to load management conditions. Load management presents the possibility of power interruption for an unknown length at unknown times. A farmer participating in a load management program realizes substantial financial reward. However, power interruption presents a scheduling problem for the farmer.

This controller could implement surge and cutback irrigation. Surge irrigation is a method that can improve irrigation efficiency. Under surge irrigation, water is applied to a field at specific intervals. Cutback irrigation reduces the amount of water flowing off the field.

This controller modified the system timing to compensate for periods of power interruption. The controller monitored the condition of power line, controlled six irrigation valves, and shut down the pump when the irrigation was completed. The controller was tested under load management conditions. The tests demonstrated that surge and cutback irrigation could be implemented even though power to the irrigation pump was interrupted.