HOW TO BUILD YOUR OWN SELF-PROGRAMMING ROBOT

BY DAVID L. HEISERMAN

2144933

621.8
H36h
Heiserman,David L.
How to build your own self-programming robot

HOW TO BUILD YOUR OWN SELF-PROGRAMMING ROBOT

Dedication

This book is dedicated to three personalities that have had the most influence on the project: my wife Judy, my son Paul, and my robot Rodney.

Other TAB books by the author:

No. 714 *Radio Astronomy for the Amateur*
No. 841 *Build Your Own Working Robot*
No. 971 *Miniprocessors: From Calculators to Computers*
No. 1101 *How to Design & Build Your Own Custom TV Games*

No. 1241
$12.95

HOW TO BUILD YOUR OWN SELF-PROGRAMMING ROBOT

BY DAVID L. HEISERMAN

TAB BOOKS
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

FIRST PRINTING—SEPTEMBER 1979

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Heiserman, David L. 1940-
How to build your own self-programming robot.

Includes index.
1. Automata. I. Title.
TJ211.H36 629.8'92 79-16855
ISBN 0-8306-9760-8
ISBN 0-8306-1241-6 pbk.

Preface

To my knowledge, this is the first practical, how-to book dealing with machine intelligence. Specifically, the book shows how to build a machine that learns to adapt to changing circumstances in its own environment. It is about a machine that programs itself to deal with problems of the moment and devise theories for dealing with similar problems in the future.

This little creature, dubbed *Rodney*, is a most remarkable machine in the present scheme of things. To be sure, there are a number of fascinating, robot-like machines roaming the workshops of amateur experimenters these days, but Rodney is in a class by himself.

Rodney is a self-programming machine, and as such, he develops a unique personality. No two Rodney machines behave exactly the same way. Wipe out the self-generated memory, and Rodney will develop another one that is bound to be somehow different from the first.

Rodney can exist quite well without any sort of direct human intervention. One exciting experiment with this machine is to wipe out its memory, roll it into a room with its battery charger, and close the door for a couple of days. Leave him alone in there for a month, if you want, but the machine won't be idle.

It will be snooping around its environment, building up its own impressions of what the world is like and developing ways to cope with it. When you finally open that door and step into the room, you will be facing a unique little personality.

Maybe you won't like certain aspects of that personality. Play with him for a while, though, and you can develop some theories on how to mold that personality into a more suitable format. Rodney is self-programmable, but he is trainable.

Building Rodney is, in itself, a valuable learning experience, but even that pales when compared with the opportunity to work with an intelligent little machine personality.

David L. Heiserman

Contents

1 Robots, Machine Intelligence, and Rodney

I would really like to know the answer to one question; why are you reading this book? I will probably never know your answer; and, at this point, you might not know the answer yourself.

It is important to determine a good reason for reading this book, because your answer will most likely determine whether or not you will finish reading it. This is a tough book to read, and if your own curiosity about real robots and your own desire to build one for yourself aren't strong enough, you are likely to become discouraged with all the technical matters crammed between these two covers.

I'm not trying to discourage you from reading the book—heaven forbid! Rather, I'm hoping to help set the stage for a big project, and get you thinking along some lines that will help you do the job successfully.

This book does not make good armchair reading. There is too much to learn, think about, and do. A casual reader is bound to get lost or discouraged early in the going. So if you are hoping to find some casual reading about robots, you would do well to look around for a different kind of book on the subject.

But, if you are honestly interested in learning something truly significant about robots, or if you are interested in getting in on the ground floor of a new technology that promises to have some impact on all our lives in the near future, this is the right book for you.

WHAT'S SO TOUGH ABOUT THIS PROJECT?

There are two phases to this project: studying and understanding the material in this book and building the Rodney robot system itself. Both phases demand a certain kind of discipline that isn't altogether common these days. While I do attempt to make it all seem as simple and logical as possible, the real burden of understanding and carrying out the instructions is on your shoulders.

The study and understanding of this book will be difficult for many readers because it demands a certain level of acquaintance with digital electronics, microprocessors and related IC devices, and microprocessor programming. I am not saying you must have a college degree in electrical engineering and computer science to understand this book; but, if you aren't acquainted with the fundamental subjects I have listed here, you will have to find a few other books to help you. Such a book would be TAB's *Understanding Electronics.*

I would suggest finding a good book describing the fundamentals of digital electronics. Such a book will help you work your way through the theory of some of the logic circuits in the Rodney system. A good book would be TAB's *Digital/Logic Electronics Handbook.*

I also suggest picking up a copy of Intel's *MCS-85 User's Manual* and Texas Instrument's *TTL Data Book* and TAB's *Digital Interfacing With An Analog World.* All are available in most computer stores. Between these three books, you will have complete descriptions of all the IC devices used in the Rodney system, and the Intel book includes a complete summary of the machine-language programming procedures.

I am doing my part of the job by presenting the theory of operation of the circuits and programming in a fashion that is as logical as possible and as complete as is practical. And, with the aids just mentioned, the rest is up to you.

Even an individual with no background in digital electronics and microprocessor can work his way through this book, the equivalent of a full one-year course in these subjects.

There is, however, a whole different angle to the matter of studying and understanding this book. Much of the theory material deals with a topic that some would call artificial intelligence. Personally, I prefer to call it *machine intelligence*, a matter of semantics we needn't deal with at this time. You are going to be studying about an intelligent machine, and that means you will be dealing with some rather new and unusual ideas. As it is applied here, there are no other sources of information on machine intelligence. In fact, other books on artificial intelligence will prove to be a hinderance to understanding this one. You will be better off knowing nothing at all about artificial intelligence, because there won't be anything to confuse the matter and you won't have to unlearn old principles that don't apply very well to Rodney. So the *less* you already know about artificial intelligence and computer programs that work with it, the easier you will understand this book. It will be tough if you are

starting out with some preconceived ideas about what artificial intelligence ought to be.

It is a good idea to read through the book first—all the way through. If nothing else, this first reading will point out what you must learn from other sources before getting really serious about the job. Then pick up any of the necessary aids and use them to help you understand the circuit theory and principles and the programming end of the job. Forget anything you might have learned about artificial intelligence, and study the book more thoroughly a second time. Only then will you be adequately prepared to start building the machine.

Building Rodney calls for some previous experience with constructing electronic circuits, especially IC circuits. You will have to know good soldering and wire-wrapping techniques, and of course, you will have to be able to follow schematics and wiring diagrams with confidence.

It isn't easy to pick up the necessary level of construction experience, something that comes only by doing it many times. Beginners would do well to find a friend or acquaintance who does have the necessary experience and can help get things going properly.

A CRITICAL TURNING POINT

Why *are* you still reading this book? Maybe you aren't so sure of the answer to that question any more. I have given you the "bad" news first, and some of you might be seriously thinking about giving up robotics for something less demanding such as fishing or gourmet cooking.

To this point you have only seen how tough the project is going to be, at least as it is filtered through the context of your own past experience. Any task this demanding must have a terrific payoff, else there would be no rational reason for carrying it any further. So, here comes the "good" news.

Rodney is a very, very special sort of machine, and before you finish reading this chapter, I hope you will see Rodney as a creature that effectively bridges the evolutionary gap between machines and animals. Just think about it—a creature that is certainly a man-made machine, and yet one that exhibits the fundamental characteristics (excepting reproductivity) of lower animals.

It is important to realize that Rodney is not merely a machine that mimics animal behavior. Researchers have been trying to build machines that mimic animal and human behavior for a long time now, and the results hardly justify all the work that must be put into the Rodney project.

Rodney is a unique creature in his own right. He often appears to behave as a simple animal, but that is incidental to the philosophy of the whole thing. Rodney belongs to a new class of machines that will soon fulfill the science-fiction fans' dream of what robots ought to be. Take a moment to consider what a real robot (*i.e.*, Rodney) ought to be.

WHAT IS A ROBOT?

The formative period of any new technology is an especially sensitive one. Mistakes or misconceptions in the early going can confuse and misdirect the efforts of well-meaning experimenters for years to come. Unless we are careful about laying the basic philosophical foundations of robotics now, we run the risk of wasting time, effort, and money developing machines and concepts that lead nowhere.

This is the time to get the basics of robotics straight; the logical starting point seems to be working up a good definition of *robot*. This is really getting down to basics, but there is a definite need for carrying an initial analysis to this extreme. The term *robot* is a coined expression that doesn't define itself as technical terms often do.

What many people think a robot should be, really isn't a robot at all. It is difficult for such people to understand the legitimate definition since they try force-fitting it to their existing misconceptions. For this reason, it is perhaps a good idea to spell out first what a robot is *not*—to tear down some old structures and make way for a more useful and exciting one.

What A Robot Isn't

There are two major classes of electromechanical contrivances making something of a stir in the popular media these days. Some of them are quite complex and very interesting machines, but they are not real robots. They are merely imitations—*parabots*, if you will.

One class of parabot calls for having a human operator manipulate the machine by remote control. Would-be roboticists must not be misled into believing any sort of remote-controlled machine that is manipulated by a human operator is any more vital to the evolution of machine technology than remote-controlled airplanes. Forget about any machine that relies on the in-line intervention of a human being. It isn't a robot.

The second major class of parabots simply replaces the remote human operator with a small computer system. It is certainly possible to play an endless variety of sterile computer programs through a cleverly interfaced set of mechanical gadgets, and end up with some fascinating effects. All this can be done, however, without really jumping the technological gap into the era of robotics.

Robot—A Matter of Semantics

When thinking about real robots, consider two alternate names, *cyborg* and *automaton*. These are the key expressions.

The term *cyborg* comes from the same root as *cybernetics*—the science of closed-loop feedback or servo systems pioneered by Norbert Wiener in the 1940s. A robot, then, must have cybernetic features, but that only expresses one aspect of how the job is done. It doesn't really say what a robot is—what separates it from any other class of automatic machinery in existence today.

Now, consider the term *automaton*. This word comes from the same root as *automatic*; what is even more meaningful is the fact it shares a common heritage with the word *autonomous*—and this word is the key to defining a robot.

A robot must be an autonomous machine, a machine capable of carrying out functions on its own. A typical computer system is not an autonomous machine. As sophisticated as some computers might be, they must interact with a human operator to do anything useful at all. A robot is not a slave, but a "free" machine. It is a free-will machine that can certainly obey the commands of a human operator, but only as long as those commands do not violate any higher-priority needs and desires.

Given a command or goal by a human operator, a true robot must be free to execute that command and achieve the goal, freely deciding exactly how to go about it. And, whenever the robot is not actively pursuing a goal set by its human operator, it must be free to determine and work toward goals of its own.

This is not a flight of fantasy, but a prime example of what a robot—an autonomous cyborg—can and must do. Any machine incapable of exhibiting autonomous behavior is not a robot at all.

Integrative Behavior Is The Key

The philosophy behind the construction of a truly autonomous cyborg, as incredible as the concept might seem at first, is not really difficult to implement these days. Buster III, described in *Build Your Own Working Robot* (TAB book No. 841), is an example of a lower-order robot. Buster III can operate without the need for direct human intervention; he can seek out his own battery charger and feed himself when the need arises; he can work his way around most kinds of physical barriers and generally interact with his environment in a fashion that clearly indicates some underlying intelligence.

Buster's brain is not a conglomeration of discrete, task-performing programs. The system is far more dynamic than is possible with the sort of thinking that goes into building parabots.

The brain of a true robot is an integrated network of simple and basic functions that are orchestrated according to on-line environmental conditions and the task set before the machine.

The Rodney system described in this book moves several steps higher on the scale of robot technology. This new machine not only reacts in a quasi-rational manner to its environment, but deals with that environment, even altering it if necessary and possible—*as judged by itself!*

HOW More Important Than WHAT

Putting together the basic working definition of a robot and the integrative techniques for implementing that definition, one central theme emerges: It is far more important at this point to think in terms of *how* a robot carries out its tasks than it is to become carried away with *what* it can or cannot do.

What really distinguishes man from other animals? Of course, we could point to an infinite variety of political, social, economic, and technological achievements through the history of mankind, but all these things merely reflect something deeper in the human makeup. The real essence of man is bound up in how his mind works, rather than what he does as a result.

Man is unique in his capacity for rational, imaginative, and often highly abstract thought. No other animal has the ability to think, judge, and interact with the environment on the same level as man does. *The thinking comes first, and the achievements are the result.*

A true robot is to other machines as man is to other animals. If roboticists can shed their current misconceptions and begin thinking in terms of an autonomous machine equipped with integrative reflex, decision-making, and goal-setting mechanisms, we can expect to see a new order of machine exhibiting a rich variety of behavioral modes that make other machines seem to be the lower-class mechanisms they really are.

IS RODNEY A REAL ROBOT?

The Rodney system is indeed an autonomous cyborg—a truly independent creature that does "its own thing." You, as Rodney's human owner, can interact with Rodney's free-will mechanisms. However, don't be surprised when Rodney treats you as a mere environmental factor, rather than an absolute master.

One of the philosophical mistakes I made when writing my book about Buster, *Build Your Own Working Robot*, was including some attractive remote control mechanisms. These mechanisms were really meant for testing purposes, but many reader/builders misin-

terpreted my intent. Rather than carrying the Buster project through to the point where he became an autonomous creature, a good many experimenters stopped at the point where they could get full control of the machine.

It certainly is exciting and a lot of fun to run Buster around the floor by manipulating a set of switches, but the scheme hardly qualifies as a real robot. It is rather an expensive, over-designed remote-controlled toy.

You will find it very difficult to play with Rodney in this fashion. Control switches are included, but mainly for programming purposes. Rodney simply cannot perform in an interesting way unless he is turned loose and allowed to pursue his own unique manner of dealing with the world around him.

At this point in the discussion, you should be getting at least an intuitive idea about what a robot should be and what Rodney is all about. By design, Rodney is an independent, free-willed creature that can operate from any one of three levels of machine intelligence.

THE EVOLUTION OF MACHINE INTELLIGENCE

Most of the so-called robots that exist today are mere shadows of what they could be. The world is primed for a major explosion in robot technology, but the people trying to light the fuse are using wet matches.

What is needed are dynamic, self-programming, general-purpose machines capable of setting and pursuing their own goals in the context of the world as the machines, themselves, perceive it. The film *Star Wars* portrays robots as free-willed machines that are fully capable of dealing with a complex environment in meaningful ways. Now that's the kind of robot people really want, and that's the kind of robot we must think about from now on.

Simple Yet Sophisticated

It is going to take some time and a lot of effort to evolve a machine as sophisticated as little Artoo-Detoo in *Star Wars*, but we have to start somewhere; I am convinced the starting point is a scheme called an *Alpha-Class robot*. An Alpha-Class robot is one whose responses are limited to basic reflex activity. One can include any number of sensory systems to sense light, sound, touch, and so on, but the responses are purely reflexive, and for the most part, random in nature.

My Buster machine includes the basic elements of an Alpha-Class robot, but Rodney more clearly exemplifies the character of an Alpha-Class mechanism. Rodney's responses include feeding (re-

charging his own battery) and a series of nine different motion patterns such as spinning clockwise or counterclockwise, moving slowly forward or backward with a slight turn, running fast forward, and so on.

Rodney's sensory system is limited to knowing when he has made contact with the feeder and sensing a stall condition when he is supposed to be moving. This machine responds to contact with the feeder by remaining motionless and absorbing energy as long as it is available. He responds to a stall condition by going through a series of quasi-random spinning and turning motions that continue until the stall condition is cleared.

Alpha-Class robots might seem too simple to be of any real importance. Indeed, they are simply little creatures, but they do not represent a trivial step in the evolution of real robots. They manage to survive quite well in a moderately complex environment just as their organic counterparts, one-celled creatures, have survived throughout earth's biological history.

Alpha-Class robots merely mark the starting point for the evolution of robot machines. A *Beta-Class robot* is slightly more intelligent than any Alpha-Class version. Beta robots have the same primitive reflex mechanisms, but they are also to *remember* reflex responses that work best under a given set of circumstances. So whenever a Beta-Class robot manages to extricate itself via a set of random responses from an undesirable environmental condition, it remembers the one response that worked and then uses it immediately whenever the same situation arises again. The responses are purely reflexive and random the first time around, but they become more rational as the machine gains experience with the world around it.

The number of sensory elements and motor responses can be extended indefinitely, but as long as the robot must encounter a given situation at least one time before learning and remembering the correct response, it remains a Beta-Class robot.

A *Gamma-Class robot* includes the reflex and memory features of the two lower-order machines, but it also has the ability to *generalize whatever it learns* through direct experience. Once a Gamma-Class robot meets and solves a particular problem, it not only remembers the solution, but generalizes that solution into a variety of similar situations not yet encountered. Such a robot need not encounter every possible situation before discovering what it is supposed to do; rather, it generalizes its first-hand responses, thereby making it possible to deal with the unexpected elements of its life more effectively.

To get a clearer impression of how a Gamma-Class robot learns and thinks, suppose its initial successful response to a bright flashing light is to attack and knock it over. The machine generalizes that particular aggressive response to a wide range of different environmental situations involving bright lights and, perhaps, any sort of light in general. It is impossible to say how deep this experience will penetrate the robot's view of his world. To be sure, this particular machine will exhibit elements of aggressive behavior that might even extend to situations other than those directly related to bright flashing lights.

Alter Its Response

Now, suppose the human experimenter sees these elements of aggressive behavior as being somewhat undesirable or inappropriate. If that is the case, the experimenter can attempt to alter the response, perhaps by flashing a light in the robot's *eyes* while speaking to it in soft tones. If the robot has learned to respond in a positive fashion to soft-spoken tones, the experimenter can alter much of what the machine deems as a threat from flashing lights. But whether or not the machine will again attack bright lights when the experimenter isn't present is something that can be determined only by careful experiment.

We are not trifling with mere gimmicks here. These are not toys or slave machines. We are dealing with the evolution of machine intelligence on a level and in terms that are rather unique today.

Why should we be content to settle for anything less than the most challenging approach to robotics?

IS THIS FOR REAL?

Are these descriptions of autonomous robots and three levels of machine intelligence for real? Or, is it all the muttering of a writer who cannot separate reality from fantasy?

Well, it's all for real, believe me. If you don't believe it, I challenge you to work your way through this book and build Rodney for yourself. If you then enter the four classes of machine-intelligence programs listed in the closing chapters of the book, and if you take the time to observe your machine's behavior carefully, you will become a believer. In fact, you'll be forced to believe it because all of this will take place right in front of you.

No, it isn't an easy task by any means. No pioneering effort of any real consequence is easy. I'm sure someone will one day start making finished, commercial versions of Rodney-like machines. Until that day arrives, however, you have a chance to do something

no other generation of man has had a chance to do—build your own intelligent creature.

HOW TO ANSWER QUESTIONS ABOUT YOUR RODNEY ROBOT

Anyone who has ever built a robot or parabot sooner or later has to answer the following question: What does your robot do? This is a terrible question because it implies your machine ought to do something that is either obviously dramatic (such as speak intelligently on some subject) or do some useful slave-type work. Most people think of robots as people-pleasing slave machines whose sole task is to either amuse or do useful work.

Unless you happen to have some acquaintances who are up-to-date as far as modern robotics and machine intelligence is concerned, you are going to be the only person around who really appreciates little Rodney. Everyone else's head is in a different place, and they are going to have a hard time understanding what Rodney is all about.

When most people find that Rodney doesn't recite poetry, sweep the floor, mix drinks, and answer the door, they are going to think you are some kind of nut. Maybe that doesn't bother you. Fine, you're in good shape.

But, maybe you don't like to be considered a nut; or more importantly, maybe you are interested in helping others understand the importance of the work. If that's the case, first try seeing the situation from the other side.

Suppose you are an inventor in the days when Thomas Edison was still setting up his first electric power company. People have heard fantastic things about electricity, but few have experienced its work first hand. And someone asks you, "What are you inventing these days."

"An electric blanket," you reply with no small amount of enthusiasm.

The questioner backs away a few steps and looks at you in a sort of funny way. After all, what kind of screwball would spend a lot of time and money working on an electric blanket that automatically rolls itself up and down on the bed?

The American public has a pretty good idea of what a robot ought to be—or at least it thinks it does. Most people draw their notions about robots from recent films and TV programs that include some mechanical critters that strike our fancy. These robots, of science-fiction tradition, certainly have some fine qualities worth obtaining, but we are still setting up the basic foundation these days.

What most people expect of a robot and what you can deliver with Rodney are worlds apart. In fact, people would be more impressed with a remote-controlled machine that could be used for fixing a good martini than a free-willed machine that actually represents a quantum jump in the technology.

It really is hard to explain Rodney and thereby justify your work to others. It takes some explaining, with you playing the role of teacher. You have to ignore the question, "What does your robot do?" and turn things around to the proper angle.

For one thing, Rodney is an adaptive mechanism. He is designed to adapt to changes in his environment. He doesn't *do* any particular task (in the ordinary sense) very well. But he is able to cope with a changing set of external circumstances better than any remote-controlled, pre-programmed machine can.

You can bind up one of Rodney's wheels, wipe out a section of main memory and destroy some of the sensory mechanisms, and Rodney will survive! Rodney will attempt to adapt his whole being to whatever conditions exist at the time. That cannot be said for some of the more razzle-dazzle parabots making the rounds on TV talk shows these days!

Rodney is the kernel of a whole new breed of creature which has never roamed the face of the earth before. Besides, if someone asks you what your robot does and they don't buy your answer, it's their problem. They are missing the whole point of a most exciting and challenging adventure.

The Rodney System —An Overall View

Chapter 1 presented a rather philosophical view of the Rodney robot system, defining what a robot ought to be and showing how the machine intelligence ought to evolve. By the time you start reading this chapter, you should have a pretty good idea about why you are engaging in the project in the first place.

This chapter provides an overall view of the system from a more technical viewpoint. It shows how a microprocessor fits into the scheme, it lists the input and output circuits you will be building, and it presents some introductory notes concerning the system programming.

A FEW COMMENTS ABOUT THE MICROPROCESSOR

The Rodney system uses Intel's 8085 8-bit microprocessor. This chip is a generation ahead of the older 8080 version that has found its way into so many modern industrial controls and consumer electronic products. The 8085 represents a marked improvement in the technology, calling for fewer output IC devices, a simpler power supply, and having a serial input/output port built within the IC.

The advantages of the 8085 microprocessor make it a compelling choice for the Rodney system. To be sure, the older 8080 costs less, at least until you add in the cost of the two major support chips and two additional regulated supply voltages.

At the time of this writing, the Intel 8085 has not yet found its way into the surplus electronics market; but, if its history follows that of all other microprocessor IC devices, it will indeed be available at a low cost on the surplus market by the time most experimenters are ready to start building Rodney.

I have not chosen to use a microprocessor simply because using microprocessors happens to be the current rage. In fact, I attempted to avoid using a microprocessor in the first on-paper designs. But, it didn't take long to discover that a microprocessor chip, especially

the Intel 8085, costs less in the long run, provides a much higher degree of system flexibility, and calls for simpler hardware designs than any alternate scheme.

After demonstrating to my own satisfaction that the Rodney system ought to be built around a microprocessor, I looked into the possibility of using some commercially available microprocessor systems, ready-made systems and bare-bones processor boards that included most of the hardware features Rodney needs. The idea was to pay a bit more money for a semi-completed system, but save a lot of time and nuts-and-bolts construction work.

The notion of fitting a ready-made home computer into the robot mainframe seemed to be an attractive idea at first. While the price of such systems is quite high, this idea has some appeal because home computers are already outfitted with keyboards and monitors that make it simple to program the system in the BASIC programming language. Plus, when the system isn't being used in the robot, it could be applied as a normal home computer that most technically inclined people want around the house anyway.

Well, that idea turned sour in a very short time. Home computers are engineered mainly for computer operations of a more traditional sort, and the problems and cost of interfacing such a system with Rodney turned out to be prohibitive. So much for the idea of fitting a ready-made home computer into the Rodney mainframe.

I had a chance to work with the idea of using a ready-made CPU board (a bare, one-board microcomputer) for quite some time. The basic boards sell for $250 or less, and they can be adapted for a wide range of control and computer applications. But these boards didn't match Rodney's special needs very well. Either there was a lot of circuitry on the ready-made boards that wasn't really needed, or else the scheme called for adding outboard circuitry that cost more than the computer board itself. So, I reluctantly scrapped the idea of building Rodney around a commercial one-board computer unit and decided to build the whole thing from scratch.

Now, I am convinced that in this particular case we are all better off building the microprocessor system from the ground up. One big advantage of doing the job this way is that it is highly educational. What better way to learn the real nitty-gritty of microprocessors than by having to work with one on a pin-for-pin basis? Of course, the entire burden of wiring the circuit and testing it falls onto the shoulders of the experimenter, but it turns out that the construction job is actually easier than the work involved in trying to retrofit a ready-made system into the Rodney scheme. Then too, there are no ICs in the system that aren't performing a vital task. Nothing is

wasted in terms of components that gobble up battery power and just go along for a ride. And finally, there has to be a certain degree of prestige associated with building a computer control system from scratch. How many people do you know who have actually done a job of that sort?

RODNEY'S MICROPROCESSOR SYSTEM

While it is a safe bet that many would-be robot builders have never tried building a microprocessor system from scratch, there is also a good chance that many have not worked with a microprocessor system in any way, shape, or form. We haven't the space in this book for describing the operation of microprocessors in general detail. Our attention will have to be directed only to those features that are relevant to the robot system. Anyone wanting to learn more about the 8085 microprocessor can glean the desired information from any of several books on the subject and, of course, Intel's own *MCS-85 User's Manual.* See *Microprocessor Cookbook* (TAB book No. 1053) for details on using the 8085 chip.

In general terms, it can be said that the microprocessor chip is a bus-oriented device. There are three major bus configurations that are adequate for all of Rodney's data gathering, processing and output operations. These three busses are named as follows: the data bus, the address bus, and the control bus.

The Data Bus

In the Intel 8085 system, the data bus is composed of eight bi-directional data lines. The chip, in other words, is capable of outputting eight bits of binary data or accepting eight bits of binary data from the outside world.

The 8-bit data words can represent program instructions the microprocessor must receive in order to know what it is supposed to do. The same 8-bit bus, however, also handles the information that the microprocessor is supposed to manipulate—sometimes accepting new information from the outside and sometimes generating information that is to be used by other devices in the system.

Where does the 8-bit programming information come from? Most often it comes from a program RAM (Random-Access Memory). The program RAM contains a list of instructions that the microprocessor is supposed to use. You will be entering these program instructions directly into the program RAM from a front-panel switch assembly. And, if you choose, the instructions can also be entered into the system from a ROM (Read-Only memory) or cassette-tape programming scheme. For the most part, however, the 8-bit instructions that enter the microprocessor via the data bus will come from the program RAM.

Remember that there are two kinds of 8-bit information running around on the data bus. The information to be processed and the results of a processing operation appear on that same bus. The 8-bit information that is to be processed by the microprocessor comes from either the program RAM or from one of the input devices. The input devices in the Rodney system include circuits that sense environmental conditions, both inside the Rodney machine and in the external world.

For example, Rodney's motors stall whenever he runs into an immovable object of some sort. This stall condition is sensed by a simple little circuit that translates some voltage signals into a binary format that is compatible with the requirements of the 8-bit data bus. These stall signals, when appropriate, serve as some information to be processed on the data bus.

There are, of course, a number of other kinds of sensing devices, but they all generate signals that ultimately become information that enters the microprocessor as data to be processed in some fashion.

Then, too, the microprocessor has to do something with the results of a processing operation. The results are dumped onto that same 8-bit data bus and sent to some meaningful place. In some instances, this worked-over data goes to a memory, but quite often it goes to some sort of output device.

Rodney's drive and steering motors, for instance, are interfaced to the data bus; whenever the microprocessor generates some information that is relevant to the operation of the motors, that information goes to the motor circuits via the 8-bit data bus.

As far as the microprocessor is concerned, the data bus is a 2-way street for 8-bit information. It can direct program instructions from the program RAM into the microprocessor, provide an input point for data the microprocessor is supposed to manipulate, and then direct the results of an operation to a memory or output device. In fact, that is the general sequence of data-handling operations for any microprocessor scheme: (1) pick up an instruction, (2) pick up the necessary data, and (3) output the results of the operation.

The actual manipulation of the information takes place within the microprocessor chip, but the data bus is the route for getting that information into and out of the chip.

Take away the data bus, and you have nothing.

The Address Bus

The address bus in a microprocessor system is often larger than the data bus, but its operation is generally simpler. Most

microprocessors on the market today have a 16-bit address bus, although the current trend in newer devices is going toward 32-bit address bus capability.

Unlike the data bus, the address bus works in only one direction. The microprocessor can place information onto the address bus, but it can never receive any information that way. The address bus is a one-direction bus—from the microprocessor chip to the outside world.

So what does the address bus do? We have already shown how the microprocessor can accept 8-bit data (on the data bus) from a program RAM. Program instructions are stored in the RAM in the order they are to be executed by the microprocessor, and there has to be some mechanism for finding the right set of data and knowing where the next set of instruction data is to come from.

Among other things, the address bus is responsible for addressing the program RAM. Or, in other words, specifying the place the next instruction comes from. When you program the Rodney system from the switches on a front panel assembly, you will specify two things with each program step: an 8-bit instruction or piece of information to be manipulated, and an address for that instruction or piece of information. So, when it is time to run the program, the microprocessor will actively specify the location of the data it needs (via the address bus), and then it accepts that information (via the data bus). The address bus is thus absolutely vital to running a program in the proper sequence.

The Rodney system also uses a scheme known as a memory-mapped I/O. This is a rather technical term that might not mean anything to you if you haven't worked with microprocessors on this level before. Rather than trying to justify the definition of the term *memory-mapped I/O*, let's go directly to the bottom line.

A system having a memory-mapped I/O is simply one that uses the address bus for specifying which one of a number of input or output devices will interact with the data bus. Just as the address bus is the route for specifying the location of a needed piece of program RAM information, it can also specify which sensing our output device is to provide or act upon data-bus information.

In summary, the address bus is a one-way street that allows the microprocessor chip to specify *where* its data-bus information is to go or from where it is to come. That's an important kind of operation.

The Intel 8085 has provisions for working with a 16-bit address bus. That many address lines figures out to more than 65-thousand locations that can be addressed. That's a lot of places for storing and fetching data. The Rodney system doesn't really need that many

address locations, so you will find only 12 lines included in the address bus.

Rodney, in other words, works from a 12-bit address bus and, of course, the usual 8-bit data bus.

The Control Bus

The data and address buses in most microprocessor systems are practically identical. They look much the same and do much the same things, although the number of lines—the bit-handling capacity—can be different and not all systems use the memory-mapped I/O that Rodney does.

The control bus schemes, however, are generally custon affairs that are tailored to a particular system. Aside from the data and address pins, the microprocessor chip has a lot of pins devoted to special control applications. Many of these pins serve output functions, telling the system that the microprocessor is accepting new information, wants new information, wants to output processed information, and so on.

At the same time, a few of the control pins on the microprocessor serve special input functions such as designating the time for starting up the whole system, entering data serially (on a one-bit-at-a-time basis), and scooting the address to some special places in the RAM.

Of all the special control functions available at the 8085 chip, the Rodney system uses only seven. These seven control functions from the microprocessor will be described in detail later on. But, as a slight preview of what is going to happen, Rodney uses control bus handling signals related to initial start up, getting serial information in and out of the system, reading and writing data, unscrambling some address and data information that appears on the same lines at one point, and signaling a read or write operation before it actually occurs.

ORIGIN OF THE BUSSES—A BLOCK DIAGRAM VIEW

In the light of the foregoing description of the data, address, and control busses, you should be able to see where and how they are interfaced with the microprocessor chip. Figure 2-1 illustrates this interfacing scheme, and it can be used for pointing out some general features common to all such systems, as well as a couple special to the 8085 and the Rodney system.

It would be nice if we could deal with the simpler parts of these schemes first, but the nature of the beast is such that little of any significance can be explained until taking on one of the more abstract

features. The case in point is the 8 lines (labeled AD0 through AD7) coming from the microprocessor chip. These designations do not appear on earlier microprocessors such as the Intel 8080. These eight lines serve two entire bussing functions at different times. Technically speaking, lines AD0 through AD7 are multiplexed bus lines—at one time they provide address-bus information, and at a later time they input or output information for the system data bus.

The multiplexing feature of these 8 lines was a necessary engineering move for the 8085 chip. It was necessary to combine addressing and data operations on the same pins in order to fit some other special functions into the 8085, functions normally carried by at least two other outboard ICs in the older 8080 version.

The multiplexing operation is carried out inside the microprocessor chip. During a typical machine cycle, the eight lower-order address bits appear on these eight lines. A moment later, this information is taken away, and the microprocessor views the same eight lines as data inputs and outputs.

So how can we separate these two vastly different kinds of bus operations? The 8085 engineers anticipated this question, and in their great wisdom, provided a special control signal dubbed ALE. You can see the ALE line in Fig. 2-1 going from the microprocessor to an 8-bit latch circuit.

The ALE line normally rests at a logic-0 level. Whenever the microprocessor is placing address information on lines AD0 through AD7, ALE goes to logic 1 and remains there as long as the address information is present.

So ALE is one of those special control signals. In this case, it indicates when the microprocessor is generating address information at AD0 through AD7. Now you will note that the ALE line is connected to the latch-control pin of the 8-bit latch. Whenever ALE goes to a logic-1 level, any information on AD0 through AD7 is entered into the latch and passed through to the address bus. When ALE returns to logic 0, that same information remains, effectively remembering what that particular address information was. When data information appears on AD0 through AD7, the 8-bit latch doesn't respond at all, because ALE is low (at logic 0) at the time. So the 8-bit latch, in conjunction with the ALE signal, is responsible for sorting out address information and data information on lines AD0 through AD7.

AD0 through AD7 from the microprocessor are also connected to an 8-bit buffer, Buffer B in Fig. 2-1. This is actually a bi-directional buffer that can pass data in two different directions at two different times. Whenever the microprocessor is generating output data, the

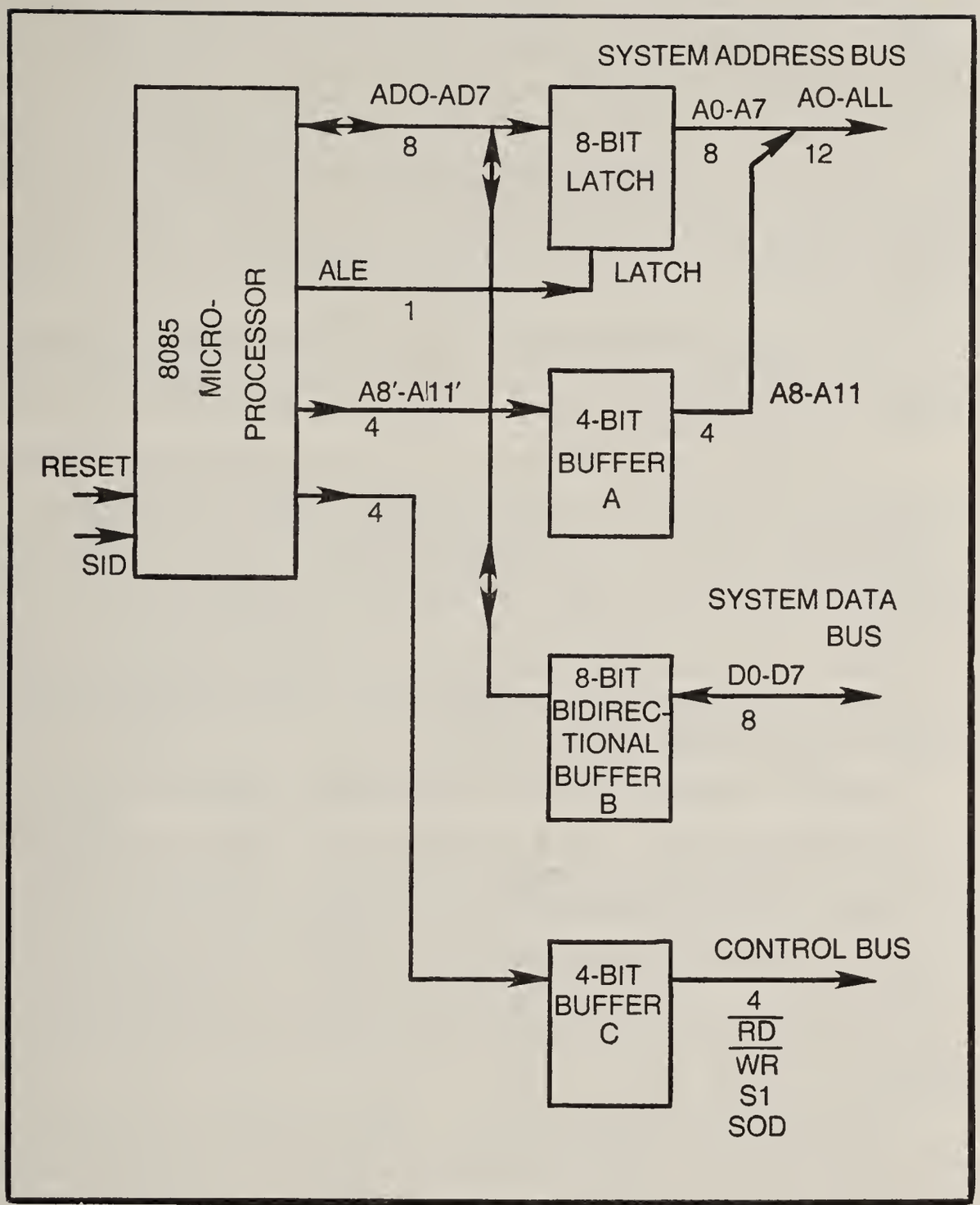


Fig. 2-1. Origins of the system busses.

data passes through Buffer B to the system data bus, labeled DO through D7. And, whenever the microprocessor is to accept data from the bus, the data passes through Buffer B in the opposite direction, from data bus lines DO through D7 to lines ADO through AD7 at the microprocessor.

An astute reader might notice that address information, during the ALE interval, will appear at the data buffer (Buffer B) as well. It is thus possible to find address information on the system data bus if the Buffer B isn't somehow disabled during the ALE interval. Rest assured, that will be done by enabling Buffer B only when the microprocessor is generating data or expecting to find data coming from the system data bus.

Actually, we haven't completely described the origin of the address-bus information. The output of the ALE-controlled latch represents the lower 8 bits of address information. The remaining 4 bits for Rodney's 12-bit address bus come from pins A′ through A11′ at the microprocessor. These four lines carry only address information. There is no data information multiplexed onto them, so there is no need for a fancy latching scheme here. A simple 4-bit buffer, Buffer A, is all that's necessary for acquiring address bits A8 through A11.

The complete address bus is thus composed of eight bits from the latch (AO through A7) and four additional, higher-order bits from Buffer A (A8 through A11).

The designations A0 through A11 for the address bus and D0 through D7 for the data bus will be carried through all of the technical discussions in this book. Becoming fully aware of their origin and significance at this time will prove invaluable for theory discussions later on.

You have already seen the ALE control line at work. Its sole purpose is to operate the 8-bit address latch, and its signal will not appear anywhere else in the system.

Another single-purpose control line is the RESET connection to the microprocessor. Pulling this pin down to logic 0 immediately stops any ongoing microprocessor activity and initializes the whole chip. You will have a RESET switch available to you at the front panel, and you will be using it whenever you start a programming or running operation. Plus, you will be hitting that switch in the event a program "blows up"—when something goes wrong and operations start going crazy.

This RESET pin (not the RESET switch on the front panel) will also be used for a power-on clearing feature. This feature will immediately initialize the microprocessor and its program whenever DC power is first applied to the system.

The four remaining control functions have a more universal application than the ALE and RESET functions do. These four control lines are buffered at Buffer C. Since it is rather difficult to define the function of these signals without some reference to the circuits they control, it would be better to wait awhile before dealing with them in a detailed fashion.

MEMORIES AND I/O PORTS ON THE DATA BUS

The foregoing section described, among other things, the origin of the system's 8-bit data bus. Now it is time to take a look at the memory devices, inputs and outputs connected to that bus.

Figure 2-2 shows the elements of Rodney's data bus system. You will see that there are three kinds of memories, 5 data input ports, and 4 data output ports. This diagram, incidentally, illustrates the fully expanded Rodney system. You will be working with a somewhat simpler system through most of the work in this book, reaching this particular point of complexity only by choice.

The presence of the program RAM should come as no surprise because we have already discussed its role to some extent. The arrows pointing in two directions from the program RAM indicate that data flows in two different directions. Under the control of the microprocessor or switches on the front panel assembly, data is either stored in the program RAM or retrieved from it at the desired moment.

The ROM (Read-Only Memory) is an optional device that can be attached to the data bus if you happen to think its advantages justify the cost and trouble involved in using it. What the ROM could do for you is eliminate the need for reprogramming the program RAM each time the system is turned off and then fired up again. The program RAM, you see, will lose all its information whenever the power supply to it is interrupted. That means taking some time to re-enter the program instructions and data from the front panel assembly.

The same kind of program information can be stored in the ROM device where it will remain, even when the power supply is interrupted. Custom ROMs, which this one would have to be, are relatively expensive and have to be initially programmed on a separate electronic system that is often dubbed a *ROM burner*. If you can come up with a good ROM device and a ROM burner, it would certainly be to your advantage to include the ROM on the system data bus scheme. Otherwise, you must live with the notion of programming the RAM yourself each time you fire up the system.

Note that the arrow at the ROM device points only toward the data bus. This indicates that the ROM cannot accept new data from the bus. It can only read out the data.

The third kind of memory in the Rodney system is a very special one we will call the system *main memory*. The main memory is a RAM-type device which, in many respects, is identical to the program RAM. The purpose of the main memory system, however, is totally different.

The main memory system is the place where Rodney stores his own knowledge of the world around him. The program RAM handles the more mundane "housekeeping" chores required for the microprocessor—and *you* program the program RAM. Rodney,

himself, is responsible for all main-memory operations. He programs the main memory system in a fashion unique to his own personality. It is highly unlikely that any two Rodney systems will have exactly the same information flowing in and out of the main memory. Note in passing that we will be referring to the main memory data with the acronym MDRL.

There are five sources of information from the outside world connected to the data bus. Sources ENVL and ENVH represent 8-bit words that characterize Rodney's environment. Taken together, the two environmental inputs make up a 16-bit data word; but since the 8085 microprocessor works only with 8-bit data words, the environment-sensing inputs have to be broken up into two groups of eight.

ENVL handles the lower eight bits and ENVH takes care of the eight higher-order bits. When it comes time to look at the environment inputs, the system generally calls for gathering information from ENVL first, followed a short time later by a call for information from the ENVH input.

Using this double-8 type of environment-sensing input, Rodney is capable of detecting more than 65-thousand variations in the world around him.

The TSWR data source is a 4-bit input port that provides the four lower-order data lines with a random number. This random number is called onto the data bus only when Rodney runs into a situation for which he has no rational responses available. The TSWR input is called a number of times when Rodney is first turned on. However, as the quantity and quality of good knowledge grows in the main memory, the significance of TSWR fades away.

The remaining input ports, labeled MARL and MARH, make up another double-8 source of data. These ports provide 16 bits of information (organized as two sets of 8 bits to make the scheme compatible with the 8-bit data bus) concerning the things Rodney had sensed in the environment prior to a new encounter with the environment. An appropriate reaction to some change in the environment is often dictated by the nature of the environment just before the new conditions arise.

Actually, MARL and MARH indicate the address location of information being retrieved or stored in Rodney's main memory. We will have more to say about this in the context of the main-memory address scheme. For now, it is sufficient to realize that MARL provides the eight lower-order bits, while MARH provides the eight higher-order bits. And, as in the case of the double-8 environment inputs, these inputs place their data onto the 8-bit data bus one byte (8-bit binary word) at a time.

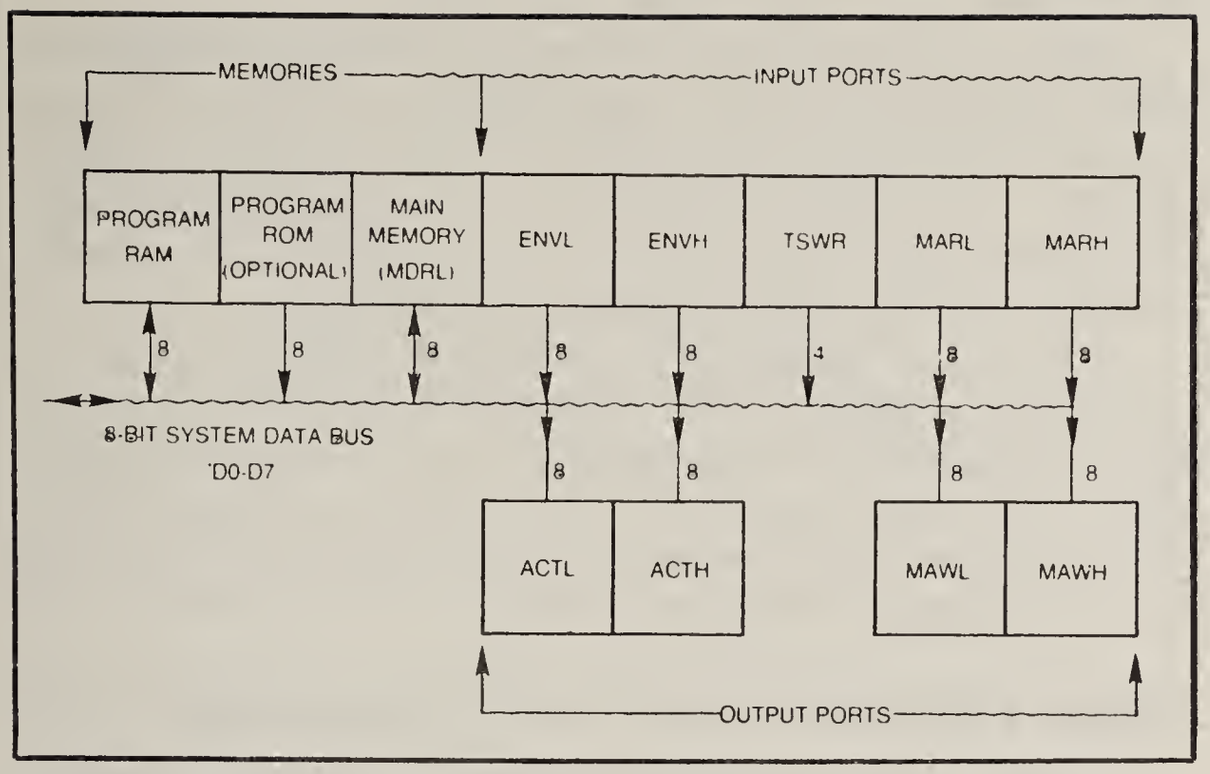


Fig. 2-2. Memories and I/O ports on the data bus.

There are only four output ports in the fully expanded Rodney system. They are all 8-bit ports that accept data on the data bus at the appropriate time, and direct that information to output-type devices.

ACTL and ACTH are "action" output ports. It is through these ports that Rodney gets information for operating his motors, blinking lights, and other things that represent his responses to changes in the environment. The ACTL and ACTH output ports complement the ENVL and ENVH input ports. Rodney senses environmental conditions through ENVL and ENVH, and he responds through ACTL and ACTH. Operated at full capacity, Rodney can respond in 65-thousand different ways.

Whenever Rodney needs to know what he did before under a particular set of environmental circumstances, he first takes a look at his main memory. If there is no viable response available for that particular situation, he looks to TSWR for a random response. And when he comes up with a response that works, he stores it in the main memory for future use at a later time.

I must repeat that the main memory is rather unique in microprocessor systems. Not many people are building microprocessor systems that have a self-programming character. In a technical sense, the unique feature of the main memory is that it isn't addressed from the system address bus. Rather, it is addressed by a pair of double-8 data sources, MAWL and MAWH.

The main memory has a 16-bit address bus. In order to work with main memory, the system has to load a pair of 8-bit latches from

the data bus. MAWL latches the eight lower-order bits and then MAWH picks up the eight higher-order bits.

When used, the 16 bits of information loaded into MAWL and MARL represent one of 65-thousand different environmental conditions that either exist at the moment or existed at some moment earlier in time. These devices serve the function of a short-term memory that can compare, moment by moment, what is actually happening with what was happening a moment ago.

Now the real meaning of the MARL and MARH data sources might seem a bit clearer to you. These are the ports that turn around the previous-moment impression of the world so it can be compared with at-the-moment impressions. What appears at outputs MAWL and MAWH can, upon command, become inputs to the data bus through MARL and MARH respectively.

MEMORIES AND FUNCTION-SELECT ON THE ADDRESS BUS

Figure 2-3 illustrates the destination of address information generated by the microprocessor. Address lines AO through A9 directly service the program RAM, selecting any one of the 1024 possible bytes of storage space at any given time. The optional ROM has a 256-byte capacity, so it is addressed with only eight lines, A0 through A7.

The function-select block in Fig. 2-3 is the hallmark of any memory-mapped I/O scheme, and it is responsible for translating the signals from five different address lines into logic levels that select any one of the twelve memory and I/O functions. These twelve functions, already discussed in connection with Fig. 2-2, are selected as needed. It is important to note that one—and only one—can be selected at any one given moment.

Just as a point of interest for the time being, address lines A10 and A11 to the function-select circuit determine whether the system will be using a memory device (the program RAM or optional ROM) or one of the I/O ports. If an I/O port is selected by the logic levels at A10 and A11, the other three address inputs, A0 through A3, become relevant. It is the combination of three logic levels on A0 through A3 that determine which one of the I/O ports will be enabled.

MORE ABOUT THE ROLE OF THE MICROPROCESSOR

The discussions thus far haven't delved very much into the internal workings of the microprocessor itself. There has been a lot of explanations about how the microprocessor generates information for the system address bus, and how it either accepts or

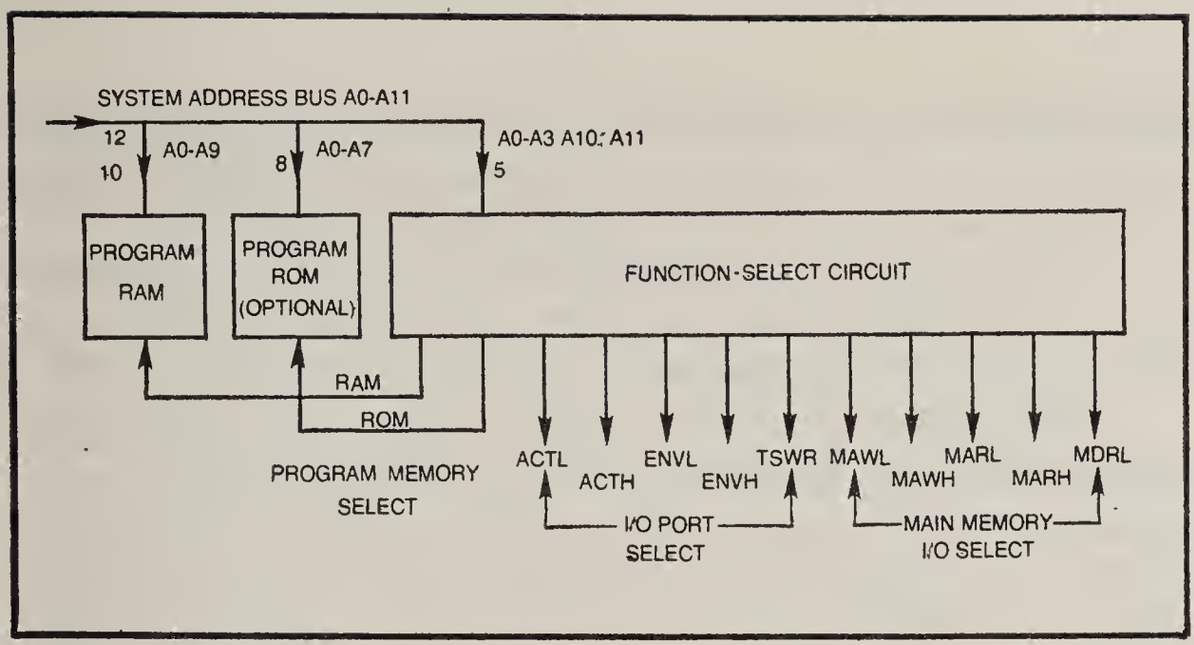


Fig. 2-3. Memories and function-select operations on the address bus.

generates data for the data bus. The control-bus operations have been included in all this, too.

Now it is time to give some consideration to the way the data is handled once it is inside the microprocessor. We won't go into a whole lot of detail, you will have to dig out much of the theory for yourself in the *MCS-85 User's Manual* and other sources of that kind.

Internal register operations are important to the job at hand, however, so let's take a look at them. The 8085 microprocessor contains a number of 8-bit registers, or storage places for 8-bit binary information. One register, the accumulator or "A" register, is of particular importance. This register can communicate directly with the data bus, either picking up an 8-bit word from the bus or generating a new one for the bus.

Most logical and arithmetic operations also take place in the accumulator register. Suppose, for instance, you want to sample the lower-order environmental conditions (ENVL) and compare them with some other 8-bit word. That "other" 8-bit word might be Rodney's idea of what the ENVL should look like, and the idea is to compare what should be with what actually exists.

The basic procedure is to pick up that "other," should-be word from somewhere, most likely from the program RAM, and place it into one of the microprocessor registers. Then, the system must be told to fetch an ENVL sample and put it into the accumulator register. Both of these bytes of information enter the microprocessor via the data bus even though they most likely come from two entirely different sources.

The system then pulls an instruction byte from the program RAM that tells it what to do with these two words. This command,

also entering the microprocessor by way of the data bus, might specify an arithmetic comparison operation, one that calls for noting whether or not the two bytes in question are equal.

After the arithmetic operation has taken place, a result signal is generated. In this particular instance, a single binary bit in a special flag register might tell whether or not the two bytes are the same.

After comparing the two bytes, it is time to take further logical action based on the result. If the two bytes in question are the same, a new instruction from the program RAM might say to ignore the whole thing and go on to something else. But if the two bytes are significantly different, yet another 8-bit instruction from the program RAM will specify what the machine is to do about it.

This sort of thing goes on and on. New data and instructions squirt into the microprocessor through the data bus, and the results of operations shoot out to appropriate destinations on the same bus. And all along, locations in the program RAM are being addressed and the function-select circuitry is picking out the devices that are supposed to be inputting and outputting data.

While the microprocessor does indeed perform logical and arithmetic operations on data, its most time consuming task is to control all the address and data traffic. A microprocessor, in itself, is a most intriguing device; only those who have an opportunity to work with it on a pin-for-pin basis can get a full appreciation of what it is doing.

All of the control operations take place as a direct result of data from the program RAM. You will be entering the programs into the RAM via a switch panel described later in the development of this project. And while it might be meaningful to introduce the programming nomenclature at this time, it is better to wait until the time arrives for trying out the ideas first hand, that is, when the system is ready to accept and execute commands.

If you find you have some spare time and are getting weary of studying this book, you might like to take a look at the 8085 instruction set—the machine-language programming notations—in the 8085 user's manual.

THE FRONT-PANEL ASSEMBLY

None of the block diagrams presented thus far in this chapter give any hint about the existance of a front panel assembly. The front panel assembly is a rather odd creature in the context of microprocessor bus systems. The front panel assembly includes twelve address switches and indicator lamps as well as eight data switches and indicator lamps. As long as the system is operating from the

microprocessor, these switches and lamps are completely disconnected from the address and data busses.
microprocessor, these switches and lamps are completely disconnected from the address and data busses.

Whenever you put the system into its PROGRAM mode, however, the microprocessor chip is completely isolated from the busses, and the front-panel switches are connected in its place.

In the PROGRAM mode, you have direct control over information appearing on the address, data, and, to a lesser extent, the control busses. This allows direct entry of programming information from the switches and to the program RAM. This feature will also give you an effective way to test all the I/O ports with the front panel, in a manner of speaking, replacing the microprocessor.

RODNEY MAINFRAME AND NEST

With all this, there is something more to Rodney than a bus-oriented, microprocessor-controlled system. There is the mechanical Rodney; there is the working machine that even the most casual observer can see and appreciate. Plus, to be consistent with the principle that a real robot has to be an autonomous creature, there has to be a nest where Rodney can absorb life-giving energy from a battery charger.

The Rodney machine, as described in this book, has a circular configuration. This particular configuration, in conjunction with a set of two identical drive and steering motors, allows Rodney to get himself out of just about any reasonable sort of physical "trap" one can imagine.

Rectangular and triangular configurations are especially troublesome when it comes to getting out of a tight corner. The sharp-angled configuration problem can be solved if the drive motors are independently controlled. However, Rodney has to be an energy-smart machine that must conserve power in every way possible. The circular configuration allows him to slip away easily from corners that would burn up a lot of power from less "slippery" designs.

Rodney has to be a battery-operated machine. That is the only practical way he can divorce himself from alternate sources of power that would restrict his freedom. Having a machine that drags a power cable along with it would be pressing the definition of robot autonomy to its very limits.

The need for operating Rodney from a battery poses some nasty problems that most roboticists would rather ignore. The worst of the situation is the simple fact that the machine, using a battery

power source that is recharged from a battery charger, has a terrible operating duty cycle. Even some of the better designs that use heavy-duty lead-acid batteries have an operating duty cycle that is no better than 1 to 4. That is, the machine has to spend 4 hours at the battery charger for each hour it runs freely around the floor.

Now that does not mean that Rodney cannot run around for more than 1 hour at a time. No indeed; the *ratio* is 1:4. Using the system specified in this book, Rodney can run around for about 4 hours at a time before needing a recharge, although to bring the batteries up to full charge again after that, he must be at the battery charger for 16 hours. All this means that an experimenter can expect to work with a free-running Rodney for about 4 hours each day. He then has to let Rodney "eat" at the nest overnight.

It is possible to learn to live with this problem of low operating duty cycles, however. One obvious solution is to have more than one battery available. One or more batteries can be standing by, with full charges, while Rodney runs around the floor. Then when his on-line battery discharges to a critical level, it can be quickly replaced with a fresh one. Be careful there—the fresh battery must be connected before the old one is removed. Otherwise there will be an interruption of power that will wipe out or scramble both the program and Rodney's main memory! At any rate, the actual hours of continuous operation can be extended to a very reasonable level in this way.

Another thought is to fast-charge the battery at the nest, dumping in the coulombs faster than they were taken out. Unless the battery is built to withstand routine fast charging, however, you are going to wind up with a bad battery in a very short time. Gel and NiCad batteries can actually overheat and explode if they are recharged faster than the specifications allow; the plates in lead-acid batteries can warp out of shape and cause internal short circuits if they are recharged too quickly. So the thought of fast-charging the battery in order to get Rodney running in the shortest possible time is a rather poor idea.

The cost is generally prohibitive, but it is intriguing to consider continually recharging the batteries from solar cells. While the solar panel would have to be very large to keep up with Rodney's normal discharge rate, the continuous trickle of current from the solar panel could extend the running time by a significant amount, especially if Rodney is allowed to play out of doors on a sunny day.

All things considered, though, a 4-hour-a-day cycle really isn't too bad. Few experimenters have that much time available to play with Rodney. Besides, it is possible to do some meaningful experiments on the system while Rodney is eating at the nest.

If you follow the recommendations in this book, you will be

building an auxiliary power supply. This is the usual, plug-in type DC power supply that is capable of operating all of Rodney's microcomputer circuitry. So while little Rodney's battery is being recharged at the nest, you can switch off the power-gobbling motors, bypass the battery system, and work with all the electronics.

You can, for instance, modify or enter new programs while Rodney is occupied at the nest. You can record his accumulated experiences from the main memory onto cassette tape and run routine checks and repairs on this mainframe and circuitry. Philosophically, a 1:4 operating duty cycle doesn't sound very attrative. In a practical sense, though, it isn't bad at all; it's all a matter of perspective.

Rodney's primary mode of expression is that of moving around the floor. Stall conditions—running into immovable objects—is the main sensory input mechanism, and moving away from such objects is the prime response mechanism.

There are other modes of expression that fit into the scheme of machine intelligence. However this one most clearly fits the definition of an autonomous creature, and of course, having the machine physically move about makes a rather dramatic case for the whole project.

Some of the alternate modes of expression include speech synthesis, optical tracking and pattern recognition, and even quasi-mechanical expressions on the face of a CRT. A machine expressing these alternate modes of expressing machine intelligence could be constructed individually, but of course they could all be incorporated into a single unit as well.

We will deal only with mechanical motion and responses to encounters with physical objects, however. The motion, in Rodney's case, is created by operating a pair of identical drive and steering motors. These are simply two independently operated motors that work together to achieve both the driving and steering operations simultaneously.

Blundering into an immovable object is sensed by looking for a stalled motor—a motor that, by all rights, should be turning, but is not. There is no need for cumbersome and tricky touch-sensing mechanisms. Whenever the motors are getting signals calling for motion, and they aren't moving, there can be little doubt that Rodney has run into something. That's how Rodney's contact-sensing works. It's simple and it's reliable.

Of course, the two-motor drive and steering mechanism must be used in conjunction with a pair of passive, freely turning wheels that serve to balance the mainframe on the two drive wheels.

Most of the electronic circuits are included on two 5 × 10-in. circuit cards. The microprocessor and buffer circuits are located on one card, while the I/O circuitry is on the second card. The two cards are interconnected by wire-wrap connections between their respective 100-pin card connectors as well as a 40-pin ribbon cable assembly.

The front panel assembly is connected to the I/O circuit board by means of three 16-pin ribbon cable assemblies. Recall that the front panel is the input point for entering housekeeping programs and testing the unit.

The circuit cards and front panel assembly are mounted on an aluminum support structure built above the battery port. The battery is mounted low to bring the structure's center of gravity close to the ground, thereby aiding the stability of the overall system. The lighter electronic components and front panel do not raise the center of gravity by any significant amount.

There is also a motor-control and power distribution panel in the Rodney system. The kinds of components required for handling the higher current levels are not consistent with low-power circuit board structures, so these heavy-duty components have to be mounted on a separate panel.

The nest is a system that is completely separate from the Rodney mainframe unit. The nest houses the battery charger, a battery current-limiting device, and the auxiliary DC power supply. Whenever Rodney makes contact with the nest, he can absorb energy from the battery charger until the charger is switched off or the batteries are fully charged.

Making contact with the nest is an accidental affair as far as the work in this book is concerned. Unlike the Buster machine described in *Build Your Own Working Robot,* Rodney does not actively seek out the nest whenever his battery voltage drops below a critical level.

Rodney could be taught to seek out his nest, but there is so much more to do with this project here that such a scheme is omitted at this time. If this range of motion is fairly restricted (to that of an average-size room, for instance), he'll manage to blunder into the nest sooner or later.

Before We Get Started

It is not easy to understand how Rodney works, it isn't easy to build the system, it isn't easy to get the machine running and programmed, and it isn't always easy to understand why Rodney does certain things. But any project of this significance is going to be tough.

I am going to help ease the burden to a great extent. My methods, however, call for some special explanation, and that's what this chapter is all about.

You will find that some of my procedures for presenting technical material are rather unorthodox at times. There is a certain kind of rationale behind these methods, and I hope you will catch the real meaning and benefit from it.

READ THE BOOK BEFORE YOU START BUILDING

You must read through this entire book at least one time before you start ordering parts and putting things together. Building Rodney and getting him running around the floor is a tremendous task, and if you don't have a complete view of what is happening every step along the way, and if you cannot see how each step fits into the overall plan, you are bound to get lost and, perhaps, discouraged. Please do not start building Rodney until you have an overall perspective on the job.

Building Rodney and getting him running demands a working knowledge of many different technical disciplines, everything from cutting and bending sheet metal to machine-language computer programming. By reading this book all the way through before you start building, you will discover your own weak points in time to do something about them.

Discovering your own weak points and seeking outside help from other people or references is a good insurance policy against costly failure. How terrible it would be to spend a lot of time and money building one phase of the Rodney project, only to run head-

long into some technical matters that bring the whole job to a sudden halt. Being prepared ahead of time, an experimenter can overcome the problem in a short time and get things underway again. Someone not prepared might face psychological and financial disaster.

Reading all the way through the book also gives you an opportunity to make some creative modifications. I would certainly not recommend making the slightest modification in the system without knowing ahead of time how everything is supposed to fit together.

I am so convinced of the necessity for reading the entire book before beginning the construction phase that I have pulled a couple of tricks. For one thing, you won't find a composite parts list. Of course, there is a parts list accompanying every schematic diagram, but I have not presented a complete list in one place. The idea here is to force you to search through the entire book, making up your own composite parts list as you go along.

You won't find a complete schematic diagram for the system either. The circuits are all here, but they are presented by sections in different chapters of the book. If you want a complete schematic diagram, you will have to search through the book and draw your own.

How much does it cost to build Rodney? How many man-hours does it take to build him? Even if I knew the answers to these questions, I wouldn't tell you. You are going to have to read the book all the way through and make a good set of notes as you go along. After that, you will have a good set of answers to these questions—probably better answers than I could provide at this time.

Some readers are bound to think these "tricks" are wholly unfair. Let me say, however, that I am doing (or *not* doing) things a certain way so that you will be better prepared to complete the job successfully.

You are mistaken if you think Rodney can be assembled like some sort of slick, commercial electronic kit. This is a *real* project that has many hidden variables that are going to cause problems for unwary and unprepared experimenters.

I cannot guarantee success for anyone unprepared for the job from the outset. I can guarantee success, however, for anyone taking the trouble to follow my advice in this chapter and play along with my "tricks."

KEEP TRACK AS YOU GO ALONG

The first item on your list of materials should be a good, thick notebook. You will need it as you make your first pass through the book, noting the parts you need and jotting down any special notes as you go along.

In fact, you should treat this notebook as a log or diary of your entire experience with the project. Keep track of where you buy your parts and how much they cost. Note when you started certain phases of the project, and keep a running commentary on all your work.

List the problems you encounter and describe your solutions in detail. Keep track of what does not work as well as the procedures that do work.

Write down whatever is on your mind, both the good things and the bad things. If you keep such a log or diary of your experiences with the Rodney project, it might one day be more meaningful to you than the book you are now reading and even the Rodney machine itself.

The Rodney project is going to occupy a significant portion of your time and thinking, and a good diary of your experiences can be something you will value for the rest of your life.

GATHERING PARTS AND MATERIALS

It is a good idea to locate and buy all the parts and materials in as short a time as possible. For one thing, you will be able to take advantage of certain price breaks for quantity ordering.

While the cost of electronic components tends to remain fairly stable, and even drop with time, the availability of microprocessor-oriented devices has been sporadic in recent years. The idea here is to "get while the getting is good." Otherwise, you might end up having to wait six months to get certain parts.

So go through the book, compiling a complete list of necessary parts and materials. You will find sources listed with most of the parts. In some instances, you ought to order from the cited source, using the specified catalog number. In other cases, use your own judgment to substitute alternate sources and devices.

You should acquire catalogs from the following sources:

Jameco Electronics, 1021 Howard Avenue, San Carlos, CA 94070
Radio Shack—any local store
Surplus Center, P.O. Box 82209, Lincoln, NE 68501
Sears—any local store

In addition to these specific sources, you will need access to a good hardware store and a firm that handles home-improvement materials.

You won't need anything out of the ordinary as far as lab equipment and tools are concerned. It would be nice if you can find a

complete sheet-metal shop and electronics lab, but most of the work can be done with simpler tools.

You will need some sort of wire-wrap tool as well as the usual assortment of tools such as a soldering iron, wire stripper, and so on. An oscilloscope with a DC input would be helpful, but not absolutely necessary. You must have a voltmeter and a reliable logic probe, however.

TEST AND TROUBLESHOOT AS YOU GO ALONG

The material in this book is presented in a step-by-step fashion. Part of the reason is to divide the task into a series of smaller, more manageable steps. There is, however, a far more compelling reason for constructing Rodney one step at a time.

Every circuit must be tested and put into good working order before the next circuit is added. Anyone deciding to build the whole project before applying power to it is bound to fail. The system is too complex to test and troubleshoot in its entirety.

So build the circuits in the order specified in the book. Test each circuit and get it working before going on to the next step. This way, if anything goes wrong, you can be reasonably certain the trouble is in the most recent part of the work.

Be honest with yourself as you go along. If something isn't working as described in this book, you must deal with it. Otherwise, that "little" problem will come back to haunt you later on.

Having to halt progress on the project to iron out a problem certainly slows things down, but it is better to spend an extra week or so dealing with troubles as they occur than to go through the frustration of trying to locate a "little" problem that messes up the whole works further down the line.

JUSTIFYING THE COST OF THE PROJECT

If you use this book as I have suggested in this chapter, you should have no trouble at all justifying the time and money you put into Rodney. Of course, you will have in your possession a truly remarkable machine creature, and there is no good reason why the project has to end with the last page in this particular book. Rodney is designed to be expanded and modified into realms limited only by your own technical know-how and imagination.

You are also buying with your money and effort an experience you will remember the rest of your life. Working with Rodney is not a trivial experience. By keeping a complete log or diary of what you do, you will be able to recapture your impressions for many years to come.

And finally, many who work on the Rodney project will reap spin-off benefits that will prove invaluable. The Rodney project can be an educational experience that can set you onto a new path in your electronics career. The things you learn as you go along apply to the most vital aspects of modern electronics technology, and anyone who follows the suggestions in these first few chapters will gain the knowledge and experience that characterizes graduates from most 2-year engineering technician programs.

Let me suggest you consider one more question before you launch this program: Are you a gutsy person? Do you have the sort of psychological makeup that is required for mastering a real challenge? If you think you are a wishy-washy sort of individual right now, you are certainly going to be a different sort of person when you finish this project! Master the challenge of the Rodney project, and you can take on anything.

What is *that* kind of education worth in terms of dollars and hours?

Robot Mainframe And Auxiliary Power Supply

The robot mainframe is the essential skeletal structure for the entire robot unit. The time you spend planning and building the mainframe has much to do with the final appearance and overall performance of the robot. The auxiliary power supply, on the other hand, seems to play a rather secondary role. It is a rather ordinary DC power supply that serves a twofold purpose: to provide a convenient and reliable source of DC power for testing and troubleshooting circuits as they are completed and, ultimately, to provide DC power to the robot while the battery is being recharged at the nest.

The only logical relationship between the mainframe and auxiliary power supply units is that they are both necessary for getting the main construction work underway. Hence the two units are described together here in a single chapter.

MAINFRAME CONSIDERATIONS

You have already had a chance to preview the essential components of the mainframe assembly in Chapter 2. In order to complete the work described in this chapter, you must be prepared to design and build the bottom plate, bottom-plate skirt, circuit card rack, and the card-rack mounting assembly.

These parts are detailed in Fig. 4-1, assuming in this case you are building the unit from 1/16-inch sheet aluminum and aluminum angle stock. The same general plan could be used if you prefer to construct the mainframe from wood or any combination of wood and aluminum.

The bottom plate is a circle of sheet aluminum with an 18-inch diameter. Since I did not happen to have access to an electric saw that could cut sheet aluminum very well, I had to resort to the following procedure. Cut or shear the bottom-plate stock to 18-inches square. Find the center of the square by lightly scribing lines between opposite corners. The point where these two lines cross is

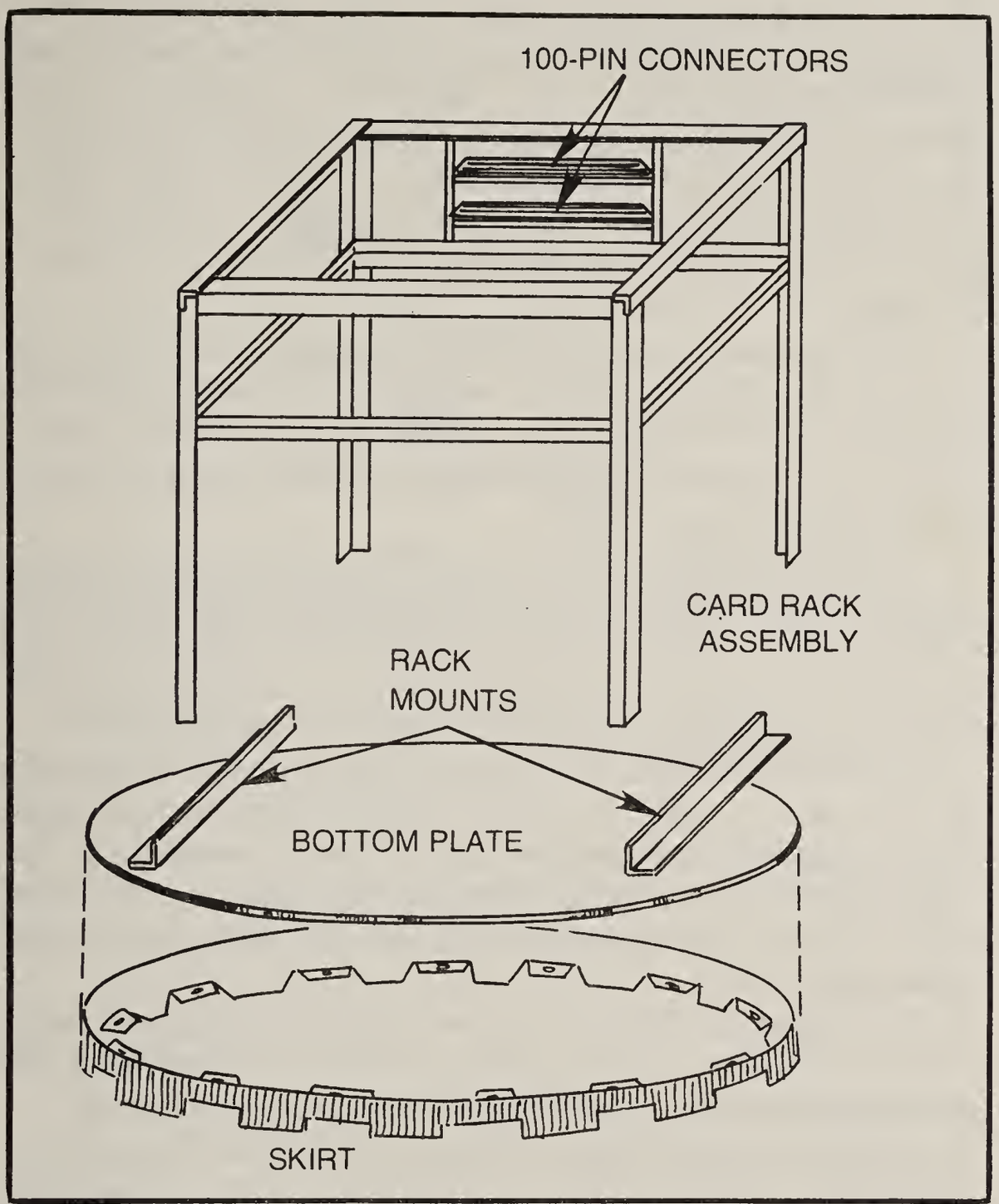


Fig. 4-1. Basic mainframe sections.

the exact center of the square. I then tied a length of string to the scribe (an ordinary 6d box nail), measured out 18 inches of string, held the free end of the string firmly at the center of the square, and scribed a neat 18-inch circle on the aluminum.

If you happen to have a saw that can cut a circle, now is the time to use it. Otherwise, a nibbling tool does a rather fine job. I simply nibbled out the circle and finished the edges with a file.

Quarter-inch plywood would work rather well for the bottom-plate assembly. The set-up procedure would be the same, of course, it would have to be sawed into the circular shape.

The bottom plate can be painted at this point. I happen to like the appearance of brushed aluminum, however, so I finished the surfaces by rubbing them with a fine crocus cloth.

Some experimenters might want to consider a bottom plate of a different diameter, perhaps a smaller one. You must bear in mind, however, that the plate has to be large enough to accommodate two circuit boards measuring 5″ × 10″. Unless these boards are situated sideways, with their long dimensions running vertically, the bottom plate has to be at least 12 inches across at the widest point. Then too, the bottom plate has to be large enough to hold the main battery, the drive motor and gear assemblies, and a relay and power distribution panel.

All things considered, the bottom plate ought to be 18 inches in diameter. The unit then won't be overly large, but there will still be enough room for mounting all the other components without everything being so crammed together that troubleshooting and routine maintenance becomes a hassle.

The bottom-plate skirt is a tricky little assembly that doesn't seem to serve a very important purpose at this point. It does, however, help strengthen the bottom-plate assembly and, more importantly, serve as a connecting point for the charging rings and any sort of outer skin you might want to attach at some later time.

The skirt is a 2-inch wide band of 1/16-inch sheet aluminum. Since the skirt is wound around the perimeter of the bottom-plate circle, its length has to be at least equal to the circumference of the circle. Using an 18-inch bottom plate, the skirt has to be cut at least 56½-inches long. Allowing a bit of slope, the skirt should be about 58 inches long.

If you have decided to use a bottom plate having a diameter other than 18 inches, simply multiply the diameter times pi (3.14) and add an extra inch or two to compensate for minor errors.

It might be hard to find a single piece of sheet aluminum that is long enough. If that's the case, simply get some 2-inch sections that add up to the desired length. Try not to use more than two sections if possible.

Now, you have a couple of lengths of sheet aluminum 2-inches wide and having a total length slightly greater than the circumference of the bottom-plate assembly. The next step is to score these bands as shown in Fig. 4-2A.

Find the center of the bands by measuring l inch from the top or bottom edge, then measure off alternate 2 inch and 3½ inch sections along the length. Begin at one end of each band and work toward the other end. You probably won't be able to get a full 2- or 3½-inch section at the opposite end of the bands, but that's no real problem. Just make sure there is at least one inch left at the ends for fastening to the bottom plate.

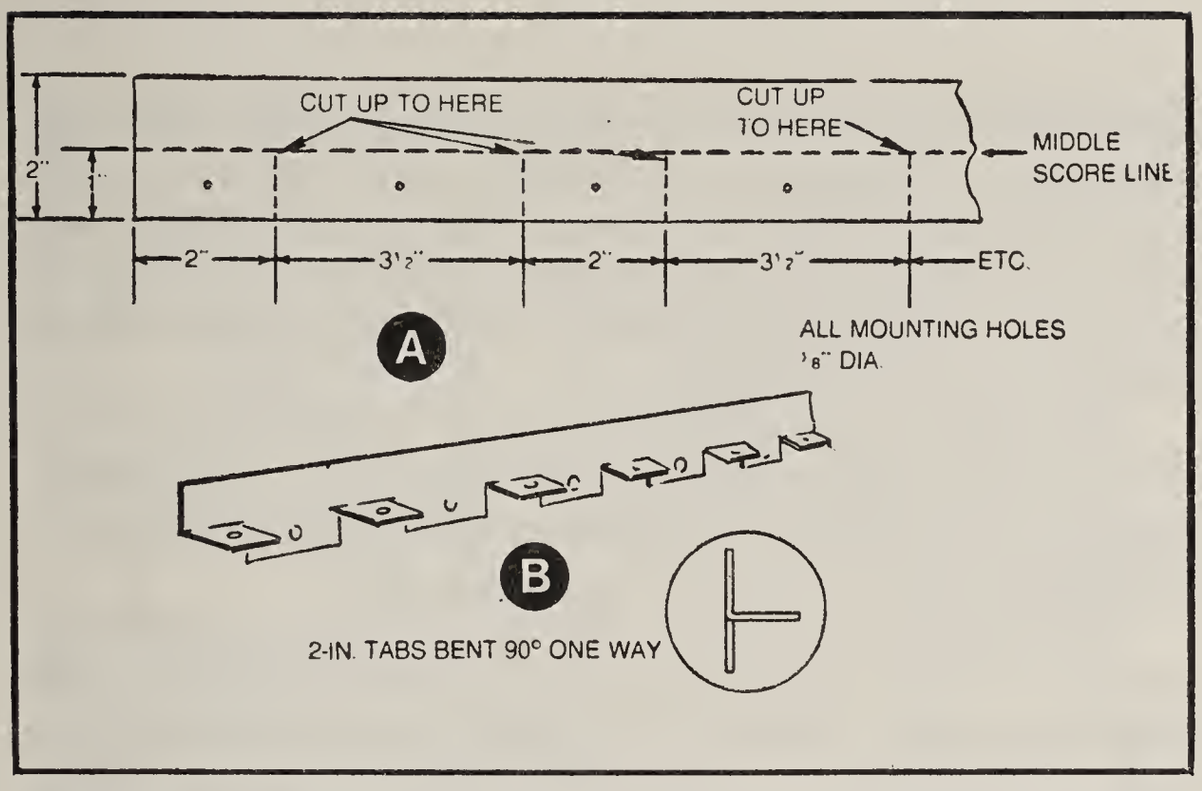


Fig. 4-2. Bottom skirt detail. (A) Scoring, drilling, and cutting detail. (B) Bending illustration.

Drill a ⅛-inch mounting hole in the center of each of the segments, and finally, separate the segments by sawing a 1-inch slot between each segment.

If you plan to finish the surface of the skirt by scrubbing it with crocus cloth, this is the time to do it. Painting, on the other hand, ought to be put off until after the next step.

Carefully bend each of the 2-inch segments at right angles to the center line. Finish the sharp corners with a file or by cutting them off with a nibbling tool.

If you wish to paint the skirt assembly, this is the time to do it.

Since my sheet-metal shop isn't really noted for producing precision parts, I had to resort to a rather crude (but workable) technique for locating the skirt mounting holes along the outer edge of the bottom plate.

The 2-inch segments of the skirt are fastened to the undersurface of the bottom plate. The idea is to wrap the skirt around the bottom plate, making a fit that is as tight as possible. I located the position of the mounting holes by simply picking up one of the skirt-assembly bands, holding it firmly in place against the bottom plate, and drilling a ⅛-inch hole right through the one already in the skirt segment and the bottom plate. That little trick might rattle the nerves of an engineering purist, but it gets the job done right and in short order.

I fastened the skirt and bottom plate together at the first set of holes with a machine screw, nut, and lockwasher. The job is fairly

routine after that, just fold the skirt assembly tightly around the bottom plate, drilling a new hole through the plate wherever the skirt is to be fastened. Fasten the assembly together after drilling each hole, and it won't be long until you are all the way around. Trim off the excess section of the skirt, fasten down the loose end, and you are done.

The card-rack assembly is the robot's superstructure. It is a structure that extends well above the bottom plate, and its primary purpose is to house the two main circuit boards and provide a mounting structure for the front panel assembly.

The height of the card rack is dictated by two considerations: the height of the battery and the vertical spacing of the two main circuit boards and front panel. The width and depth of this unit is determined only by the 5- ×10-inch dimensions of the circuit boards.

My own card-rack assembly is 15½-inches tall, allowing a generous 8 inches for the battery to slip in under the circuit boards. The circuit boards and front panel can then be stacked in a 7½-inch space above the battery compartment. The rack has an outline width of 11 inches and is 8 inches deep.

Figure 4-3 shows the general construction of the card-rack assembly. There is plenty of latitude here for applying some creative construction methods of your own. Just keep in mind several key points. First, the battery assembly must fit through the bottom of the rack. Then too, there has to be at least 1½ inches (preferably 2 inches) of vertical space for each of the circuit boards and front panel.

I assembled my own card rack from a combination of ¼-inch and ½-inch aluminum angle stock. This material is generally available in 6- or 8-foot sections at hardware stores. I used ordinary machine screws for fastening the sections together; but when I get a chance to rebuild the unit, I will probably invest in a rivet tool and fasten the structure together that way.

Aside from making certain there is adequate clearance for stacking the two circuit boards and front-panel assembly, another critical dimension is the placement of the two short sections of angle stock that hold the card sockets. The standard mounting holes spacing for 100-pin card sockets is 6-¾ inches. It would be a good idea to doublecheck that dimension against the spacing of the holes in your own card socket. If you don't have the socket available yet, forget about attaching the mounting angles into place until you do have the socket.

In fact, you might note another little problem concerning the placement of the two socket-mounting angles—the socket is not exactly centered on the 8800-series Vector boards. It turns out that

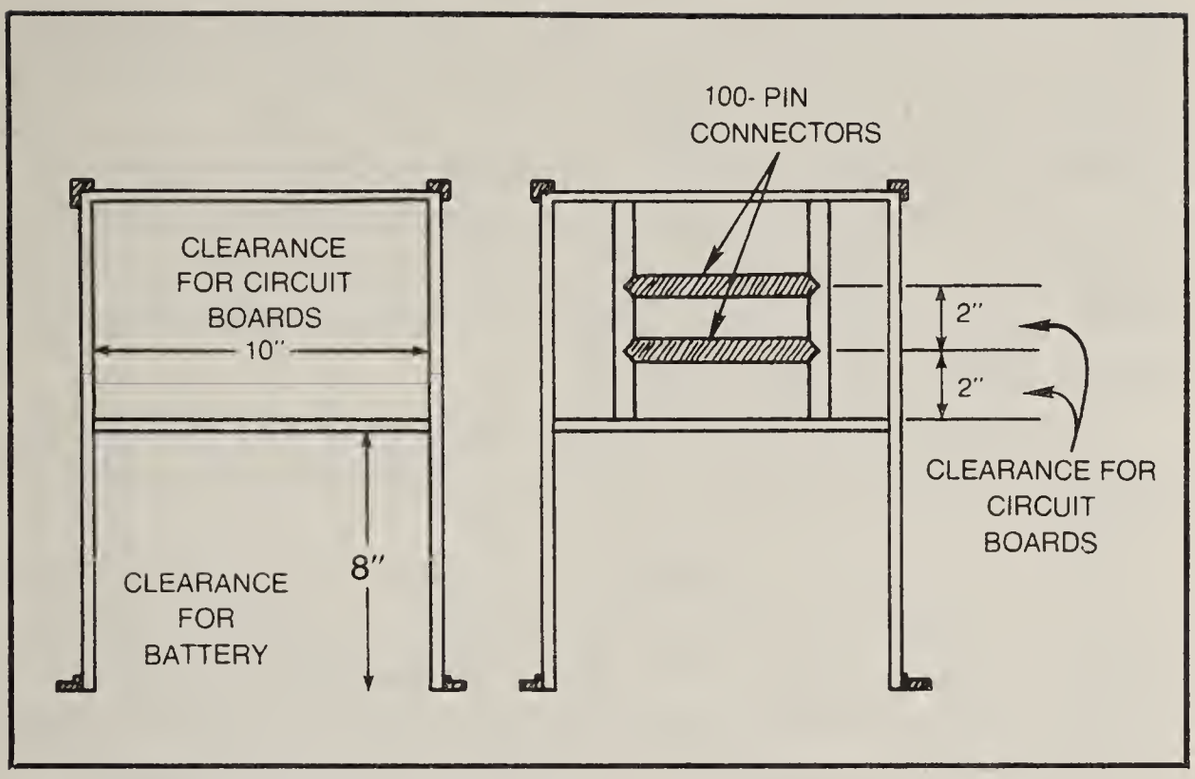


Fig. 4-3. Card rack assembly drawing.

the 100-pin connector is 1-½ inches from one end of the circuit board and 2-⅛ inches from the other. So unless you have both the socket and card at hand, there is no point in installing the socket-mounting pieces.

The card rack is fastened to the bottom-plate assembly by means of a pair of aluminum angles. The inside-dimension spacing of these angles should be equal to the outer-width dimension of the card-rack assembly (11 inches in my own case), but they ought to be as long as possible. These two rack-mounting angles, also add a great deal of rigidity to the bottom plate, rigidity that will be necessary under the stress caused by the torque action of the drive motors and the weight of the battery.

See the photo in Fig. 4-5 for an overall impression of how the mainframe assembly should look at this point in the construction. If you have in mind any variations of this recommended layout, it is important to study the rest of the book so that you can anticipate any problems that might arise later in the project.

AUXILIARY POWER SUPPLY

The auxiliary power supply is a rather ordinary DC power supply that provides up to 5A at about +12V and 1A at +5V. The +5V output is regulated, but the +12V output is not.

The +12V output will be used for testing the circuit boards, relay assemblies and the motors. Ultimately, this 12V section will supply power to the robot's electronics while the battery is being recharged from a standard battery charger.

While the +5V section of the auxiliary power supply will not play a vital role in the finished system, it is quite helpful for checking out some of the circuits as they are built. No matter how careful an engineer and technician might be, there are bound to be some construction errors now and then. As suggested earlier in this book, it is a good idea to check each circuit or sub-assembly before it is committed to the final system. The +5V source from the auxiliary power supply will thus prove helpful in making certain things are working properly as they develop.

The complete schematic diagram for the auxiliary power supply is shown in Fig. 4-4, and the recommended parts list is detailed in Table 4-1.

For anyone who has built a DC power supply before, the construction of the auxiliary power supply is a rather straightforward project. A single 12V, 5A power transformer (T1) and a full-wave bridge rectifier assembly (BR1) supply full-wave rectified DC for both the unregulated +12V and regulated +5V outputs.

The input to the bridge rectifier is fused to protect the bridge assembly. The unit is protected from short circuits at the +12V output by fuse F2. The +5V output is not fused for the simple reason that the 5-volt regulator (VR1) has a built-in overcurrent-protection feature.

Table 4-1. Parts list for the auxiliary power supply.

CS1—8-ft., 3-conductor cordset (Jameco 17238 flat)
S1—SPST toggle switch (RS #275-602)
L1—120V neon lamp assembly (RS #272-705)
T1—12V, 5A filament transformer (RS #273-1513)
Fa, F2—5A slow-blow fuse (RS #270-1287)
BR1—6A, 50V full-wave bridge rectifier (RS #276-1180)
C1—1000μF, 25WVDC electrolytic capacitor (Jameco)
C2—1000μF, 16 WVDC electrolytic capacitor (Jameco)
VR1—5V, 1A regulator; 7805 or LM340T-S (Jameco)

J1, J2, J3, J4—2 pr. red and black binding posts (RS #274-662)
TS-1, TS-2—2 sets of 2-terminal barrier strips (RS #274-656)
Regulator heat sink (Jameco 291-.36H)
2 ea. fuseholder (Jameco HKP)
Enclosure (RS #270-270)

Suppliers:

RS = Radio Shack

Jameco = Jameco Electronics
1020 Howard Ave.
San Carlos CA 94070

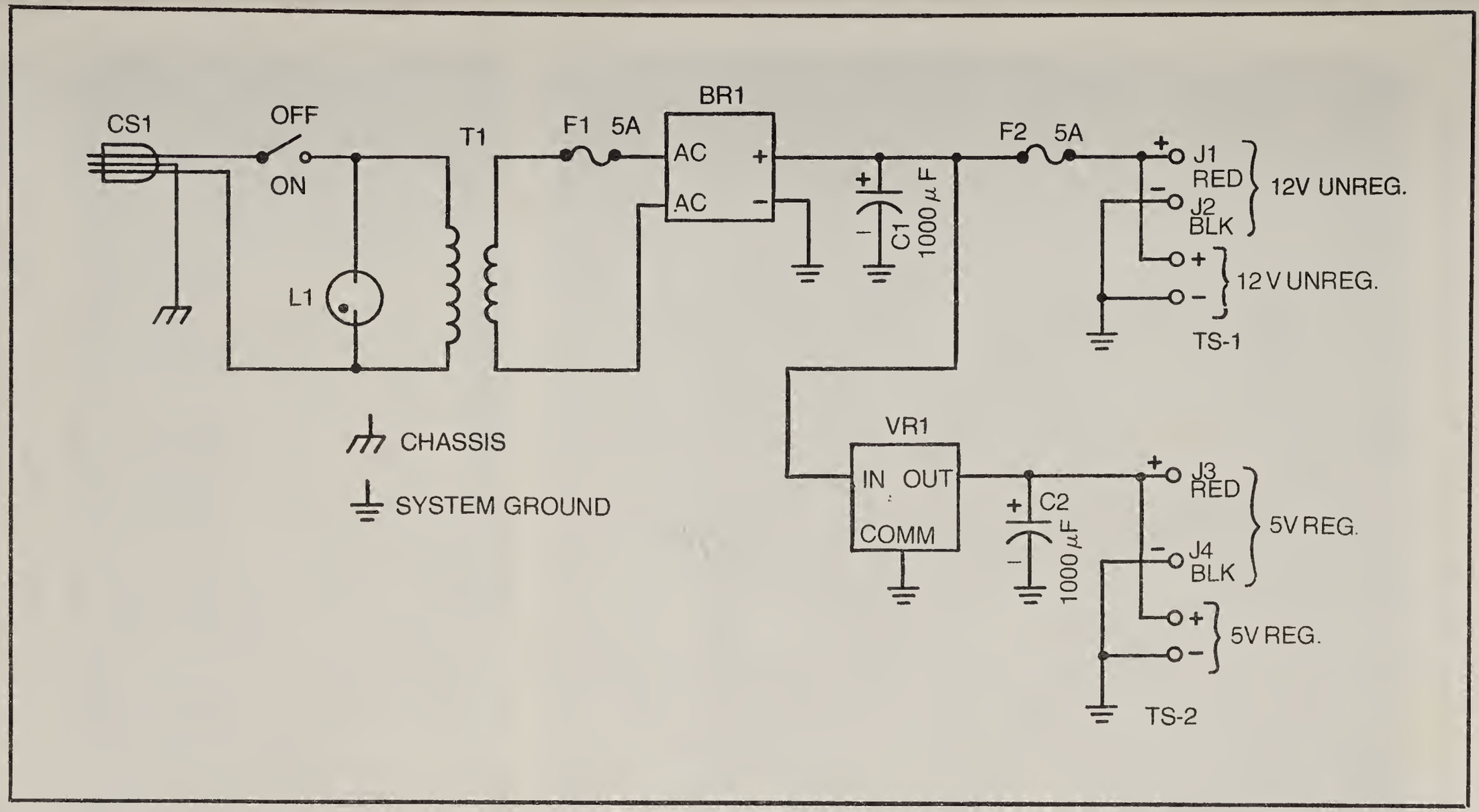


Fig. 4-4. Schematic diagram of the auxiliary power supply.

Fig. 4-5. Rodney mainframe assembly.

The entire unit is switched on and off by means of S1, and the neon lamp assembly (L1) is simply an ON/OFF indicator lamp. The cordset (CS1) ought to be a 3-conductor, flat cable that is as long as possible. This will not be an important feature during the construction phases of the project, but it will pay off later on when Rodney begins running around the floor.

Recall that the auxiliary power supply will eventually become an integral part of the nest assembly. And since the nest should be located near the center of Rodney's play space, it follows that the cordset has to be long enough to reach the nest and flat so that Rodney can run over the cord without stalling.

As far as basic construction and testing procedures are concerned, start by installing the cordset, ON/OFF switch, indicator lamp and power transformer. Plug in the unit and turn it on. The lamp should go on and off in response to changes in the setting of the ON/OFF switch.

Measure the AC voltage at the secondary of T1. When the unit is turned on, the voltage should read in the neighborhood of 12.6 Volts. Any discrepencies at this point should be ironed out before going any further.

Install and wire fuse F1, and screw the rectifier assembly (BR1) to the bottom of the chassis. Connect filter capacitor C1, turn on the unit, and measure the DC voltage across C1. The voltage at this point should read about 18V DC—positive at the (+) terminal of C1.

If you happen to have a 10-watt resistor of 2.5 ohms to 10 ohms, connect it across C1. This operation will check the current-drive capability of the circuit. If things are working properly, the voltage across that resistor might drop as low as 10 to 15 volts, and the resistor will get pretty hot. Don't run the circuit very long this way,

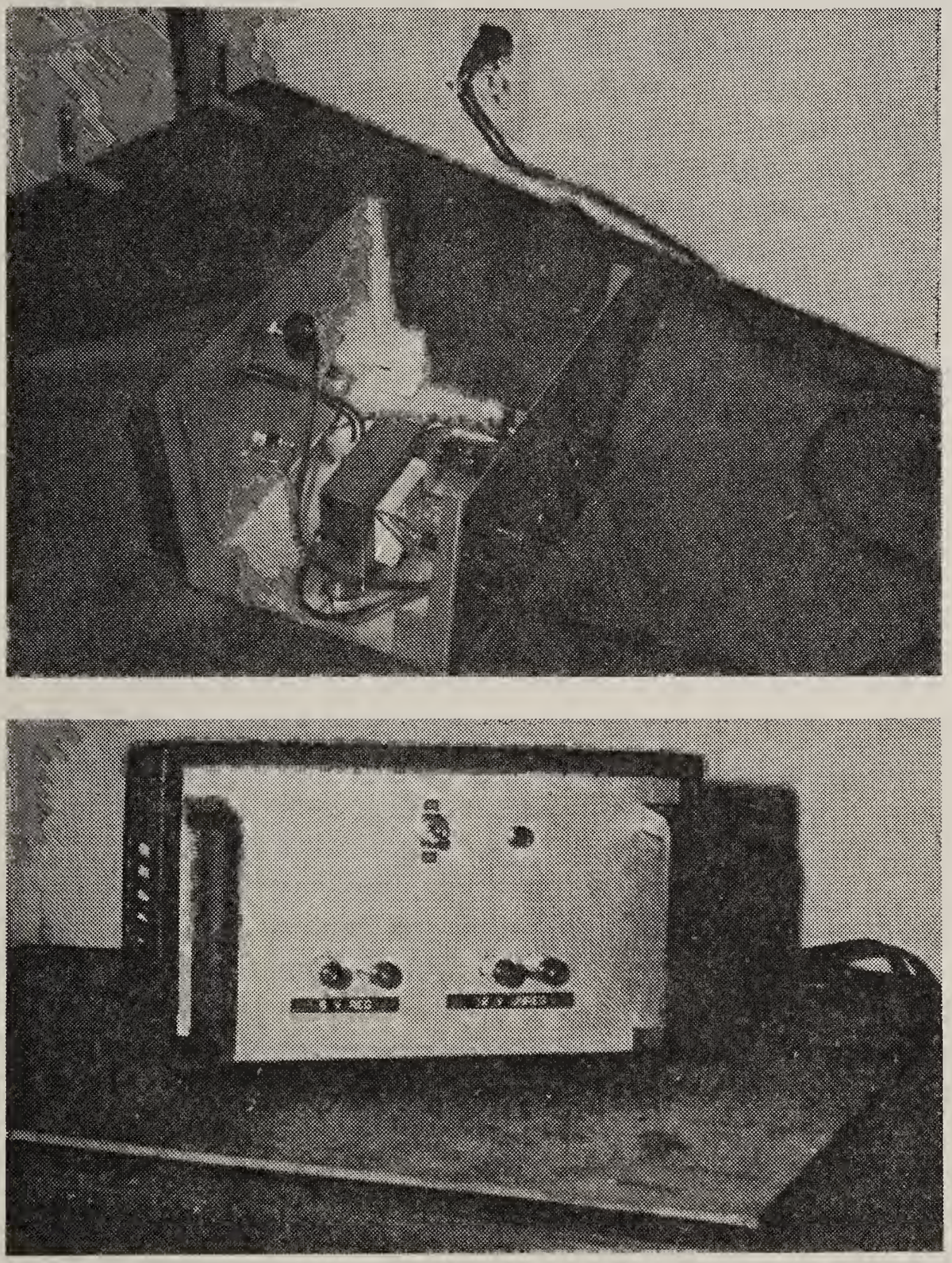


Fig. 4-6. Construction photos of the auxiliary power supply. (A) Rear and inside view. (B) Front view of the completed unit.

just long enough to make sure T1 can supply the current and F1 holds up under the load.

Connect the positive side of C1 through F2 and to both a red 5-way binding post (J1) and one connection on a 2-terminal barrier strip, TS-1. Connect the negative side of C1 (or the circuit ground) to the black 5-way post (J2) and the second screw terminal on TS-1.

With the load resistor used for earlier tests removed, turn on the power supply and check the voltages at J1 and J2 and then at the two sets of screw terminals on TS-1. Both should read about +18V DC.

Attach the 5V regulator assembly (VR1) and recommended heat sink to the bottom of the chassis. Connect the input terminal of VR1 to the positive terminal of C1, and wire the COMM connection of VR1 to the system ground (to J2, for example). Turn on the power supply and measure the voltage at the output terminal of VR1. You should see between 4.5 and 5.5 volts DC at this point.

Complete the circuit by wiring C2, J3, J4 and TS-2 as shown on the schematic in Figure 4-4. Doublecheck the wiring by measuring the DC voltage between J3 and J4, and then across the two sections of barrier strip TS-2. You should see 4.5 to 5.5 V DC at both test points.

Front Panel Program/Test Assembly

While Rodney is essentially a self-programming machine, there is still a need for entering a small selection of programs from the outside world. An animal might be born knowing exactly nothing about how to cope with its environment, but even so, it is born with an underlying reflex mechanism that helps the learning process begin.

As far as the experimenter is concerned, there is a need to give Rodney some rudimentary reflex mechanisms in the form of simple computer programs. Please understand that these programs do not dictate what Rodney should do under a given set of circumstances. Rather, the programs tell Rodney he should do something—it is up to the machine-intelligence feature to determine what Rodney does.

All of the rudimentary reflex programs are initially entered into the Rodney system via a switch-and-lamp panel assembly. This front panel is thus vital to any robot operations because it is the primary mechanism for entering the machine-language programs.

As you work your way through this book, you will find that the front panel is also your primary testing and troubleshooting aid. You will be using it to check out some of the more complex circuits as they are constructed and installed.

The front panel, the subject of this chapter, thus serves two purposes; it is both the means for entering basic reflex programs for the machine intelligence, and it is a good test and troubleshooting aid for the entire system.

As indicated in Figure 5-1, there are two sub-systems involved in building, testing, and troubleshooting the front panel assembly. First, there is the front panel, itself. This unit consists of 12 address switches and indicator lamps, 8 data switches and indicator lamps, and 3 control switches. The second part of the front panel operations includes a portion of the I/O board (often called Board 200 in this

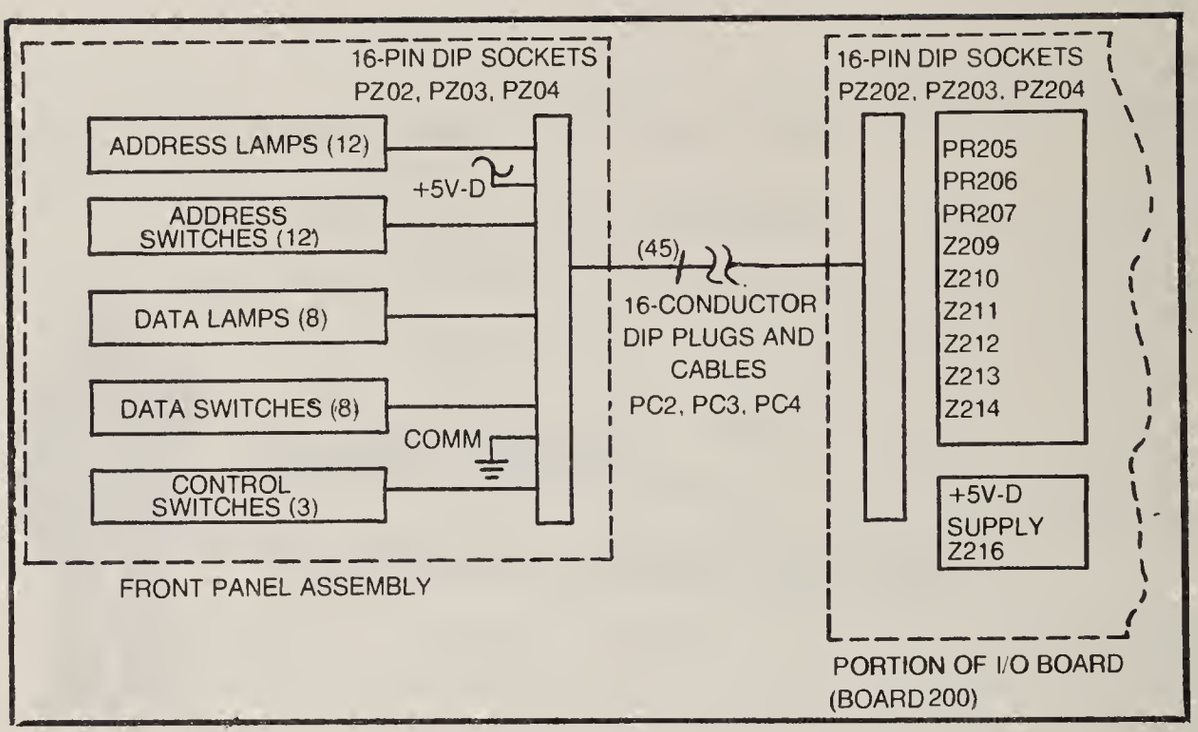


Fig. 5-1. General layout of the panel system.

book). The I/O board is one of the two main circuit boards in this system.

In this chapter, you will see how to assemble the front panel in its entirety, but you will be constructing only a portion of the I/O board, the portion relevant to front panel operations. You will probably want to mount the front panel on the top of the card-rack assembly that has already been described in Chapter 4. The I/O board is the upper card in the card-rack assembly. The front panel and I/O board are interconnected by means of three 16-pin ribbon cable and plug assemblies.

It would be a good idea to study through this chapter carefully before you begin gathering the parts and putting things together. Make some notes of your own as you study through this material, and try to draw at least a mental picture of how the system looks as the work progresses. If nothing else, study the parts lists in Tables 5-1 and 5-2. You don't want to be caught short of any parts midway through the job.

THEORY OF OPERATION

Technically speaking, the purpose of the address and data switches is to give the operator direct access to the system's main address and data busses. When these switches are being used, the 1 or 0 logic levels are placed on the respective system bus.

When it is time to write reflex information into the program RAM, for instance, the address switches directly address the program RAM. At the same time, the data switches can provide 8-bit data directly to the data inputs of the program RAM.

When the front panel is in use, the address and data lamps indicate the logic levels on their respective busses. The address lamps show the system address, and the data lamps indicate the 8-bit data on the system's data bus. It is therefore possible to read or write data directly at the system's program RAM. As an added bonus, the front panel switches and lamps can also deal with any of the I/O ports you will be installing at a later time.

Even before the microprocessor chip itself is installed, you will be able to use the front panel system for I/O operations such as running the motors, sensing the condition of the environment, and so on. This is the testing feature of the front panel system.

Since the front panel switches and lamps interact directly with the system's address and data busses, the microprocessor and front panel assembly should never be enabled at the same time. If, for example, the microprocessor is putting data and address information onto the busses at the same time the front panel is doing the same thing, there is going to be a devastating conflict. The system is operated either from the front panel or from the microprocessor chip—never, never both at the same time.

Much of the front-panel circuitry included on the I/O board is devoted to making certain these two sources of bus information are never on the line at the same time. The key to separating the front panel and microprocessor operations is a simple toggle switch located on the front panel. This switch is labeled PROGRAM/RUN, and its signal is designated $R/\overline{P}$ on the schematic diagrams.

Whenever this switch is in its PROGRAM position, the microprocessor is completely isolated from the system's address and data busses. For all practical purposes, the microprocessor chip isn't even in the circuit. In fact, the PROGRAM mode makes it possible to do some relevant test operations without having a microprocessor anywhere in the system. You will be taking advantage of the PROGRAM mode before completing the work in this chapter, even if you don't have the microprocessor chip yet.

To get a more detailed impression of how the PROGRAM/RUN modes work, look at the block diagrams in Fig. 5-2. The address switches (ASO through AS11) in Figure 5-2A go to the system's 12-bit address but (A0 through A11), but through a pair of 3-state hex buffers (Z209 and Z210). Whether or not the address information from the address switches passes through the buffers and to the system address bus depends on the logic level present at the $R/\overline{P}$ connection. If this control connection is at logic 0, for example, the system is in its PROGRAM mode, and the information at the twelve address switches passes through the buffers and onto the address bus.

Setting the RUN/PROGRAM switch to its RUN position, however, sets the R/$\overline{P}$ signal at logic 1, and the buffers (Z209 and Z210) are placed into their high-impedance output state. No matter what sort of information is present at the address switches, none of it can pass through the buffers to the address bus. When the microprocessor is installed, you will find that only the microprocessor can send address information to the address bus in the RUN mode.

The 12 address lamps shown in Fig. 5-2A merely indicate the status of the address switches. These lamps indicate the panel address status whether the system is in its PROGRAM or RUN mode. They cannot show the status of the address bus during a RUN operation. (At microprocessor speeds, the address lamps would be useless anyway). The logic inverters (Z211 and Z212) invert the logic levels to provide an adequate, active-low drive current for the LEDs.

In short, the panel address switches provide bus address information only when the system is set for its PROGRAM mode. The address lamps indicate and confirm the status of the panel address switches.

Incidentally, an experimenter finding himself short of cash for the project can omit all twelve address lamps and the logic inverters without causing any real problems. In such a case, one merely has to be doubly careful about setting the address switches to the right positions before committing any data to the data bus.

The operation of the data portion of the front panel assembly is somewhat more involved. As indicated by the arrows in Fig. 5-2B, the data-switch assembly is bi-directional—that is, the circuit is capable of both putting information onto the data bus and retrieving it from that same bus.

As in the case of the addressing circuit, the data switches have access to the system data bus only when the RUN/PROGRAM switch is in its PROGRAM position. Whenever the system is set to its RUN mode, R/$\overline{P}$ is a logic 1 and the transceiver buffers (Z213 and Z214) isolate both the data switches and data lamps from the system data bus. Microprocessor operations can then take place on the data bus without conflict.

Setting the system to its PROGRAM mode, however, changes R/$\overline{P}$ to logic 0, and as indicated in Fig. 5-2B, enables the data bus transceiver. Eight-bit data can now be placed onto the system data bus from the data switches, *or* the data lamps can respond to data present on the bus from some other source. (That "other" source does not include the microprocessor, however. Remember that the

microprocessor is isolated from the system during a PROGRAM operation).

So the data portion of the front panel can either deliver or accept 8-bit data at the system data bus. However, it cannot put information onto the bus and accept data from the bus at the same time! So how can we separate these two panel data operations?

Data reading and writing operations from the front panel are separated from one another by means of the panel's LOAD pushbutton. The logic level from this pushbutton is designated LOAD throughout this book. The bar across the LOAD connection in Fig. 5-2B indicates it is active low. That is to say, it is a logic-0 level

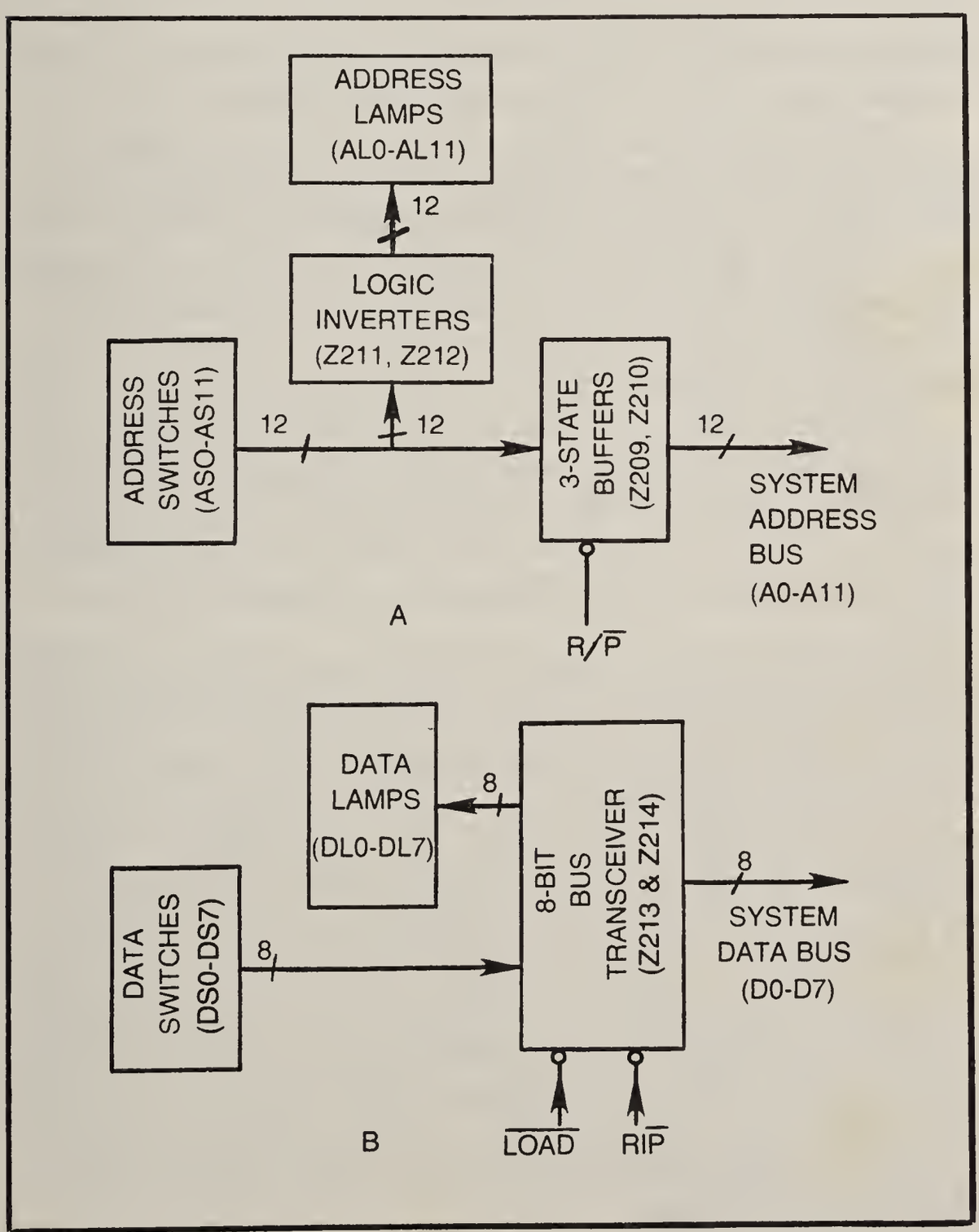


Fig. 5-2. Panel block diagrams. (A) Address switch and lamp section. (B) Data switch and lamp section.

whenever the system is set for LOAD (the LOAD pushbutton is depressed), and is a logic-1 level whenever the LOAD button is not depressed.

Pressing the LOAD pushbutton sets LOAD to logic 0, and data is allowed to pass from the eight data switches onto the system data bus. Releasing the LOAD pushbutton reverses the data operation, allowing 8-bit data from the system data bus to register on the eight data lamps.

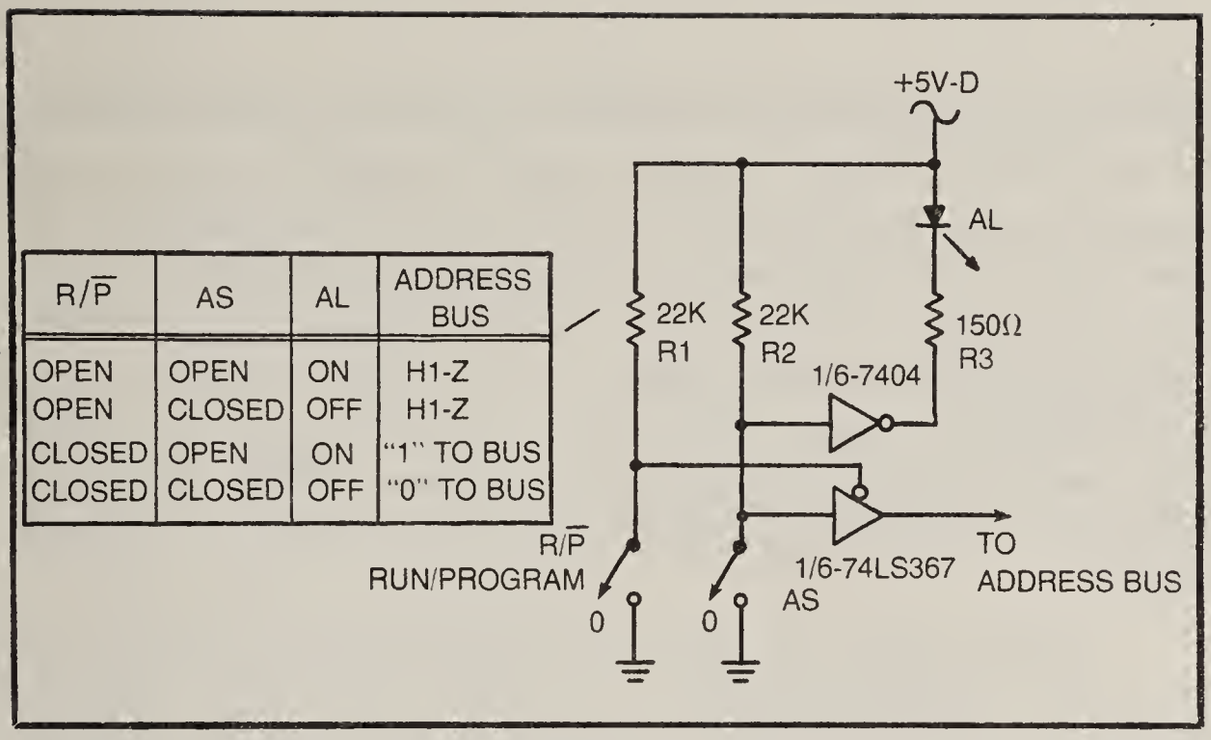
In summary, it is possible to place 8-bit data from the data switches onto the system data bus by doing two things: setting up the system for its PROGRAM mode ($R/\overline{P} = 0$) and depressing the LOAD pushbutton ($\overline{LOAD}=0$). Eight-bit data is then read from the data bus and applied to the data lamps when the system is set up for its PROGRAM mode and the LOAD pushbutton is *not* depressed. Setting the system to its RUN mode isolates both the data switches and data lamps from anything going on in the system.

Like the addressing scheme, the data switches and lamps can be operated independent of any microprocessor operations. You will be using the data switches and lamps to check the operation of the system RAM and all of the I/O devices, long before the microprocessor chip is installed. But while the address lamps can be omitted without having any adverse effect on the system, the data lamps must be included.

Now, let's take a much more detailed look at the circuitry for the panel address and data scheme. Rather than attempting to sort through the complete circuit, we'll look at isolated, representative examples. Then if you understand the isolated examples, you'll be well on your way to understanding the front-panel system as a whole.

Figure 5-3 represents one of the 12 identical address circuits in the front-panel assembly. The figure shows one of the address input switches (AS), the address indicator lamp (AL) for that same switch, the RUN/PROGRAM toggle switch, and the associated logic inverter and buffer device. Everything in this circuit, with the notable exception of the RUN/PROGRAM switch and its 22k pull-up resistor (R1), is duplicated twelve times in the front-panel circuitry.

Note that the address switch (AS) is connected to the address bus through one section of a 74LS367 3-state buffer. The 3-state control line for this buffer is connected to the RUN/PROGRAM switch; as long as that switch is open, resistor R1 pulls up the switch connection to near +5V, or logic 1. A logic-1 level at the 3-state control of the buffer effectively turns it off so that AS has no effect at all on the address bus.



R/P	AS	AL	ADDRESS BUS
OPEN	OPEN	ON	H1-Z
OPEN	CLOSED	OFF	H1-Z
CLOSED	OPEN	ON	"1" TO BUS
CLOSED	CLOSED	OFF	"0" TO BUS

Fig. 5-3. Simplified circuit for one of twelve address switch and lamp circuits.

Closing the RUN/PROGRAM switch, on the other hand, sets the 3-state control terminal of the buffer to COMM potential, or logic 0, and effectively opens the buffer so that any logic level from AS appears on its respective line of the system address bus.

Whenever switch AS is closed, its output point is pulled down to logic 0 by means of the ground connection. Opening switch AS breaks that ground connection, and resistor R2 pulls up the output connection of the switch to a logic-1 level. These 1 or 0 levels from switch AS are fed directly to the appropriate line on the system address bus as long as RUN/PROGRAM is in its logic-0, PROGRAM position.

The address indicator lamp for this particular address switch is labeled AL in Fig. 5-3. The anode of the LED is connected to the +5V power supply, and thus turns on only when its cathode is pulled down close to COMM or logic 0. Now suppose address switch AS is in its "1" position. That means it is generating a logic-1 address bit. By digital convention, this logic-1 signal should cause the corresponding LED indicator to light. (Logic-0s are normally indicated by a lamp that is *not* lit). So this logic-1 level from AS goes through a simple logic inverter, one section of a 7404, where it is changed to a logic-0, the level required for lighting AL. In short, setting AS to its "1" position causes AL to light up, confirming a logic-1 setting. Resistor R3 in series with AL is simply a current-limiting resistor that prevents excess current being drawn from the inverter.

Closing switch AS so that it is generating a logic-0 level causes the 7404 inverter to output the opposite level, namely a logic 1. Since logic-1 levels are very close to +5V, AL does not light up. It

can't light up because it has very close to the same potential on both its anode and cathode. Lamp AL simply registers the logic level being generated at any time by switch AS. The really critical part of the circuit is the 3-state buffer that allows AS information to reach the system address bus only when the RUN/PROGRAM switch is in its PROGRAM (logic-0) position.

The little circuit in Fig. 5-4 illustrates the operation of a panel data switch and lamp circuit. In this instance, all but the LOAD switch and resistor R1 are actually repeated eight times on the panel circuit.

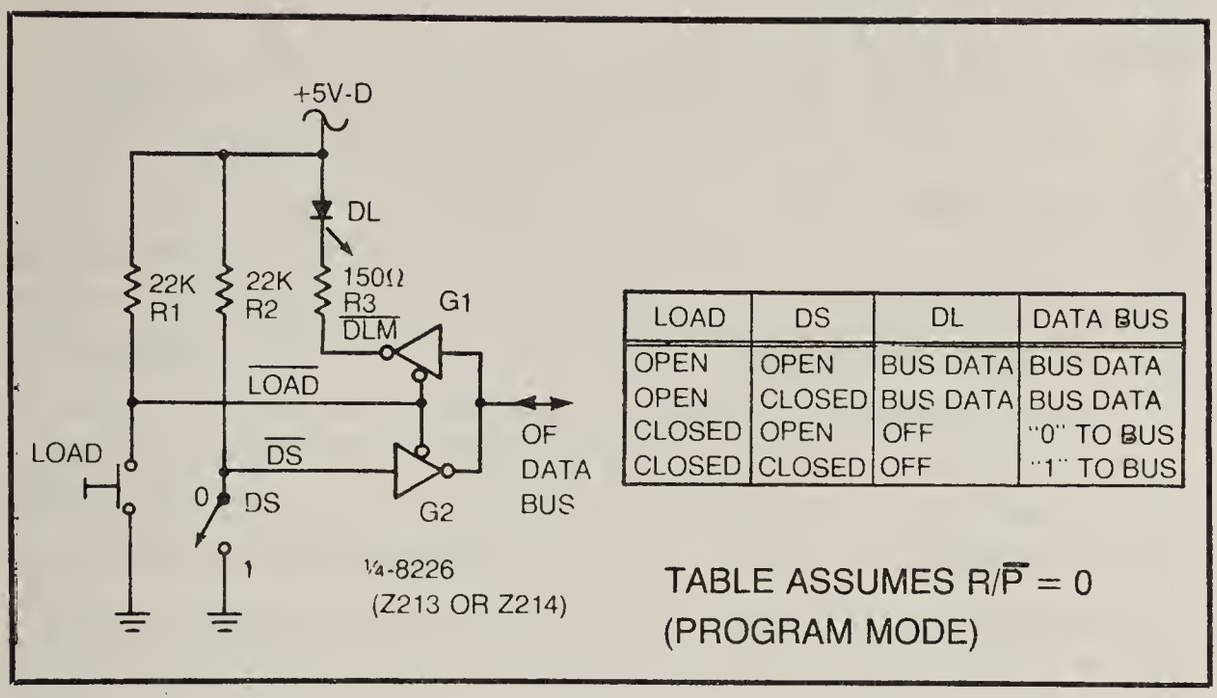
The data switch DS generates logic-1 and logic-0 levels, depending on whether it is open or closed. When this switch is closed, it actually generates a logic-0 level by virtue of the fact that this pulls its output connection down to COMM level; when the switch is opened, the COMM connection is broken open, and resistor R2 pulls up the output level of the switch near +5V, or logic 1. Note, however, that the switch appears to be labeled backwards. The labels for DS show a logic 1 when the switch is closed, and a logic 0 when the switch is open. Isn't there some sort of contradiction here? Is there an error on the diagram?

The answer to both questions is a simple "no". At least not if you follow things all the way through. Notice that the output of data switch DS goes through an inverting 3-state buffer to one line of the system data bus. This means the logic level from the data switch is inverted before it reaches the data bus. Therefore, the labels on the data switch indicate the logic level that actually reaches the data bus—and that's what is important anyway. Really, you shouldn't care less what the switch itself is generating as long as you know what is going to the data bus.

Whenever the DS data switch is set to its "1" position it does indeed deliver a logic-1 level to the data bus. Likewise, setting DS to its "0" position causes it to send a logic-0 level to the data bus. Of course, this all assumes inverting gate G2 is enabled.

According to the diagram in Fig. 5-4, G2 is enabled whenever a logic-0 level is applied to its active-low enabling connection. This connection is wired directly to the LOAD pushbutton on the front panel; so whether the data from DS is sent to the data bus or blocked by G2 depends on the setting of the LOAD switch.

LOAD is wired in such a way that its output is a logic-1 level whenever the button is *not* depressed. The switch contact is open in this instance, and resistor R1 pulls up the LOAD switch level to logic 1. Depressing the LOAD switch grounds its output connection, thereby making it generate a logic-0 level.



LOAD	DS	DL	DATA BUS
OPEN	OPEN	BUS DATA	BUS DATA
OPEN	CLOSED	BUS DATA	BUS DATA
CLOSED	OPEN	OFF	"0" TO BUS
CLOSED	CLOSED	OFF	"1" TO BUS

Fig. 5-4. Simplified circuit for one of eight data switch and lamp circuits.

As long as the LOAD switch is open, a logic-1 level is delivered to the enabling pin on G2, switching the buffer to its high-impedance state; switch DS is now isolated from the data bus. Depressing the LOAD pushbutton enables the buffer so that an inverted version of the logic levels from DS can reach the data bus. Putting it rather simply, a 0 or 1 logic level is sent from the data switch to the system data bus whenever you depress the LOAD pushbutton.

Next, take a look at the corresponding data indicator lamp, DL. Unlike the address circuits, this data lamp is not operated from the corresponding data switch. Rather, it is operated from logic levels on the data bus. Follow the flow of information from the data bus to the indicator lamp.

Assuming for the moment that 3-state buffer G1 is somehow enabled (you guess how), you can see that any logic level present on the data bus line will be inverted as it passes through G1 and operates DL. If the information on the data bus line happens to be a logic-1 level, it will emerge from G1 as a logic-0 level, effectively grounding the cathode of DL, causing the lamp to turn on. If G1 is enabled, a logic 1 on the data bus causes DL to turn on. Everything seems to be in order so far.

If the data on the data bus happens to be a logic 0, this level will emerge from G1 as a logic 1 which, in turn, causes DL to turn off. Data lamp DL thus mimics the logic level present on its line of the system data bus, assuming, however, that G1 is enabled.

Have you made any good guesses about how G1 is enabled so that DL can respond to data bus information? The LOAD switch is the key to the matter. As described earlier in this discussion, the output from the LOAD switch is at logic 1 as long as the LOAD

switch is *not* depressed. This logic-1 level is connected to the active-high enabling pin of G1, turning on the buffer as long as the LOAD button is *not* depressed.

In other words, data lamp DL registers data bus information as long as the LOAD button is not depressed. Depressing the LOAD pushbutton disables the indicator lamp circuit, but enables the switch circuit. So the switch and lamp circuits work 180 degrees out of phase. Depressing the LOAD button sends switch information to the data bus, and releasing the LOAD button allows the data lamp to read whatever data is on the data bus from some other source.

For the sake of simplicity, the RUN/PROGRAM circuitry is not included in Fig. 5-4. Actually it will be present, making it impossible to put information onto the bus or read information from it as long as the system is in its RUN mode. The circuit in Fig. 5-4 affects the data bus (or is affected by that bus) only when the system is set to its PROGRAM mode.

ELECTRICAL COMPONENTS OF THE FRONT-PANEL SYSTEM

Now that you have had an opportunity to consider the finer details concerning the operation of the front-panel circuitry, it is time to take a tour through the complete circuit. If you've had some trouble understanding the theory of operation to this point, I'm afraid you won't get a whole lot out of this discussion.

In any event, give this section some careful attention, because, if nothing else, it will help you understand how and why some of the circuit components are set up.

Figure 5-5 shows the complete schematic of the address switches, address lamps, and control switches as they appear on the front-panel assembly. Bear in mind from the circuit in Fig. 5-3 that there are also some pull-up resistors, limiting resistors, buffers, and inverters included in the address circuitry. All of these components, however, are located on the I/O circuit board. What we are discussing here is the wiring of the front panel.

The twelve address switches (labeled ASO through AS11) all have one terminal connected directly to COMM. The second terminal on each of these switches goes to a pin on one of two different 16-pin DIP sockets, PZ02 and PZ03.

The three control switches: LOAD, RUN/PROGRAM, and RESET, are connected in an identical fashion, one end of each going to COMM and the other ends going to pins available on DIP socket PZ02. Of these three control switches, RESET has not yet been mentioned. Its purpose is to initialize the microprocessor; since we are a long way from installing the microprocessor, there is no need to

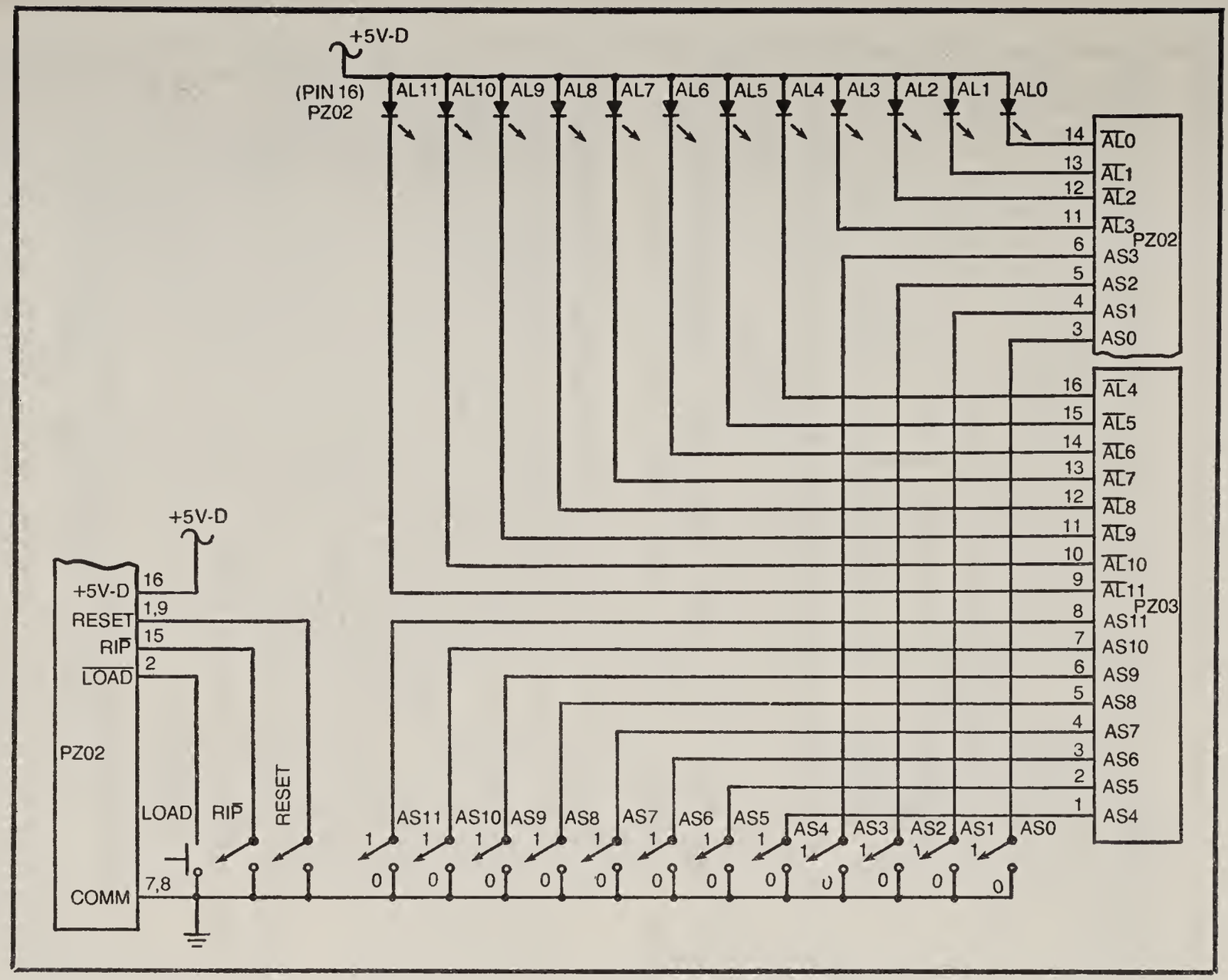


Fig. 5-5. Complete schematic diagram for the front-panel address switches, address lamps, control switches, and associated plug connectors.

discuss this switch in much detail. Simply install the RESET switch as described, then it will be available later on when you need it.

The anodes of all twelve address indicator lamps (labeled AL0 through AL11) are connected together to the +5V-D power source. Each of the cathodes is connected to a pin on sockets PZ02 or PZ03.

As mentioned earlier in a rather brief fashion, PZ02 and PZ03 are 16-pin DIP IC sockets. The lamps and switches shown in Fig. 5-5 are wire wrapped to the designated pins on these sockets—the cathode of AL0 goes to pin 14 of PZ02, the output side of AS0 goes to pin 3 of PZ02, etc. ICs are not plugged into these sockets, however. Rather, they are connecting points for 16-pin ribbon cable and plug assemblies that interconnect components mounted on the front panel with those essential components on the I/O board.

So let's see where they go.

Fig. 5-6 shows the components of the address circuitry that are located on the I/O board. The connections between the front panel and the I/O board are picked up at PZ202 and PZ203 in Fig. 5-6. These PZ numbers represent plug connections PZ02 and PZ03, respectively, from the front-panel plugs and cables. The reason for the slight change in designations is that all components mounted on the I/O board have a 200-series designation.

Referring back the Fig. 5-3, each one of the address switches should have a pull-up resistor to +5V-D. This fact is reflected in the overall address schematic in Fig. 5-6 by component PR207.

This component, however, requires some special explanation. In a sense, it is a home-made, 24-pin IC device. Actually, it is a 24-pin DIP header that can hold up to 12 ¼-watt resistors. Since you need 12 pull-up resistors for the twelve address switches, the DIP-header trick works out quite nicely. The schematic for this PR207 resistor component, as well as another used for the 150-ohm limiting resistors, can be found in Fig. 5-7.

Twelve different pins on PZ202 and PZ203 bring the address switch connections to the I/O board, with the resistor header component (PR207) providing a 22k pull-up resistor for each of these connections.

The address switch in Fig. 5-3 also goes to the input of an inverter. In Fig. 5-6, the twelve necessary inverter connections are made at Z211 and Z212. After the address switch signal passes through the inverter, it is supposed to go through a 150-ohm resistor. That's the role of a second resistor-header device, PR208. (See Fig. 5-7 for details of PR208).

After passing through the 150-ohm current-limiting resistors, the address switch signals then return to the cathodes of the address

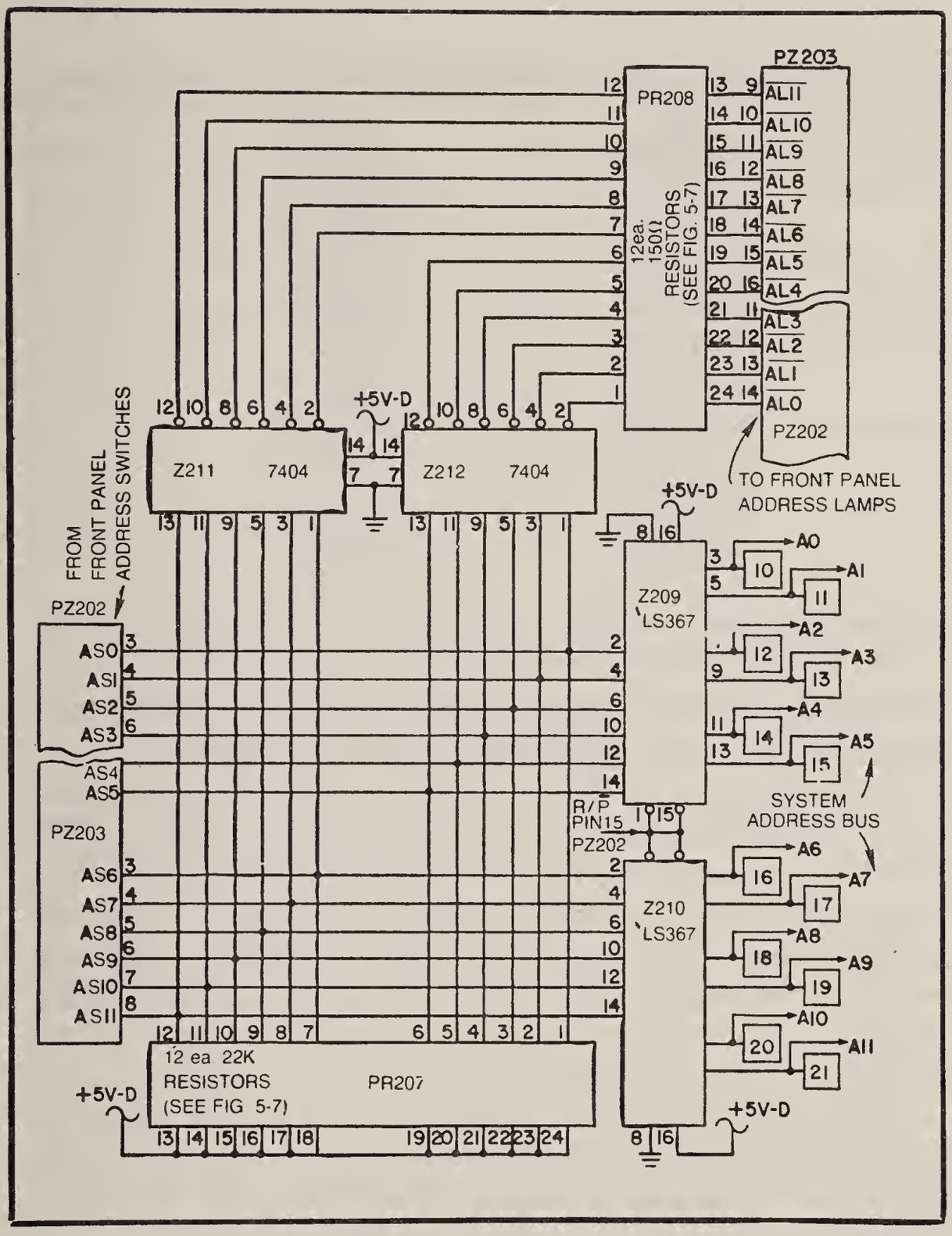


Fig. 5-6. Complete schematic diagram for the I/O board portion of the panel address switches, lamps, resistor networks, and plug assemblies.

indicator lamps on the front-panel assembly. This feat is accomplished, once again, by the two 16-pin DIP plug-and-socket arrangements.

But the address switch signals go to yet another place, somewhere besides a set of pull-up resistors and logic inverters. They also go to 3-state buffers connected between the switch connections and the system address bus.

This vital set of connections is represented by ICs Z209 and Z210 in Fig. 5-6. These are the hex 3-state buffers. Their inputs are the same points that go to pull-up resistors (PR207) and inverters

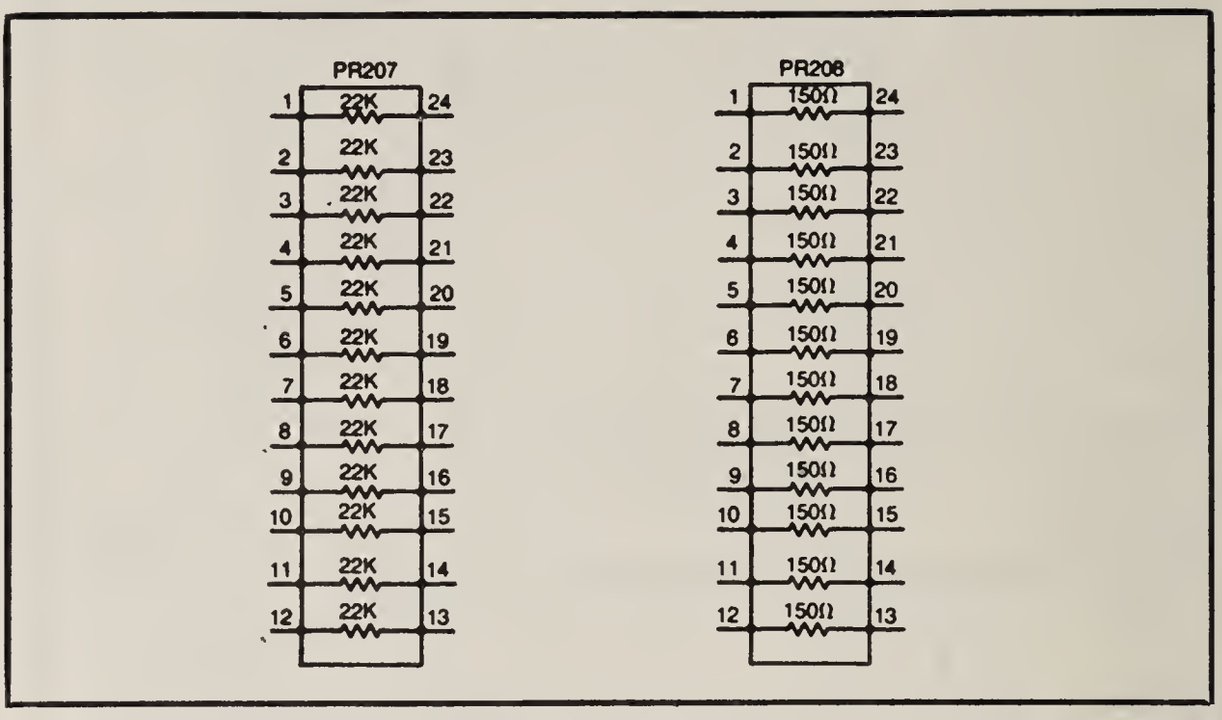


Fig. 5-7. Schematic and top-view layout of resistor headers PR207 and PR208.

(Z211 and Z212). The outputs of these buffers go to the system address bus, as indicated by designations A0 through A11.

Incidentally, this is the first time we have encountered a schematic convention that will be used through this book. The address bus connections in Fig. 5-6 are shown by little squares and others with arrows. The squares indicate connections to the main plug connections on the circuit card. This I/O card has 100 such connections that will ultimately slip into a compatible 100-pin connector socket. The numerals written in the squares indicate the pin numbers. More on that later.

The address designations noted by arrows indicate wire-wrap connections elsewhere on the same circuit board. All of the address-bus connections in this case go both to the card's plug connections and elsewhere to other circuitry on the board.

The 3-state buffers in the addressing circuitry are supposed to determine whether or not address-switch information passes to the address bus. As described earlier in this section, this is the $R/\overline{P}$ signal function. You can see the $R/\overline{P}$ signal connected to two places on Z209 and Z210. This completes the analysis of the overall addressing scheme.

The 8-bit data circuitry follows much the same path as the address circuits. The data switches are located on the front-panel assembly; a plug-and-cable assembly carries the data-switch signals to the I/O board where the buffers and resistors are located, with the cables carrying the lamp information back to the data lamps on the front panel.

Refer to the simplified version of the data circuitry in Fig. 5-3 while working your way through this analysis.

Data operations begin with the eight data switches (DSO through DS7) in Fig. 5-8. Closing any one of these switches causes a logic-0 level to appear at the corresponding terminal of plug socket PZ04. A 16-pin DIP ribbon cable assembly carries these data-switch logic levels to plug PZ204 on the I/O board.

The data-switch portion of the I/O board is shown in Fig. 5-9. Here you can see PZ204 introducing the DS signals to two different kinds of devices, a 24-pin DIP resistor header (PR205) and a pair of inverting bus transceivers (Z213 and Z214).

As shown in Fig. 5-10, PR205 is another one of those home-made "ICs" composed of 22k pull-up resistors. As far as the data-switch circuitry is concerned, these resistors pull up the switch levels to logic 1 whenever the switches are open.

Z213 is a quad, inverting bus transceiver that can either pass $\overline{DSO}$ through $\overline{DS3}$ signals to the data bus (DO through D3), or accept signals D0 through D3 from the bus sending them to the 150-ohm current limiting resistors on PR206. Recall that the LOAD pushbutton determines the direction of data flow, and note the $\overline{LOAD}$ signal going to the direction control pin on Z213, pin 15. This pin determines whether switch data goes to the data bus or data-bus informa-

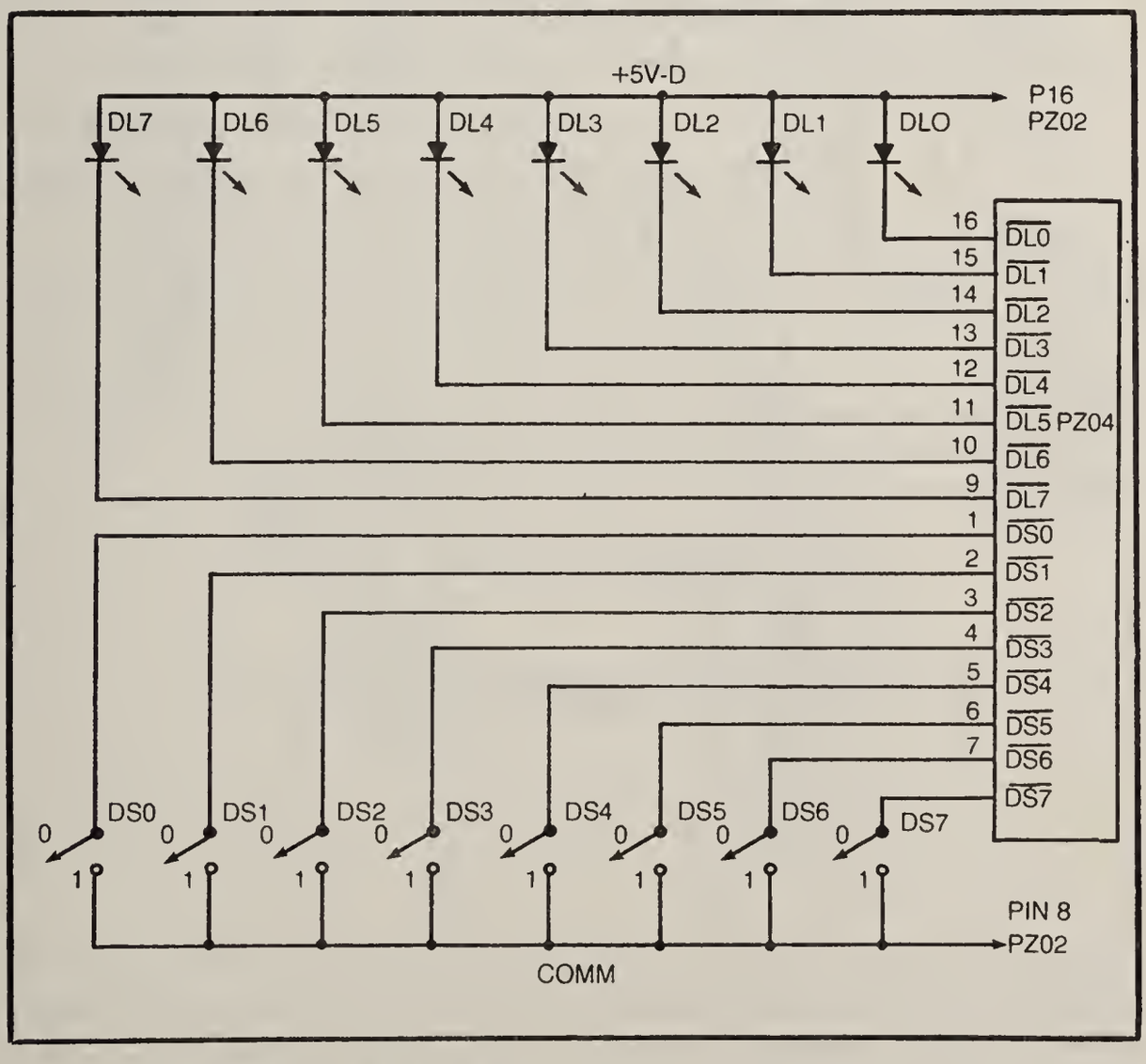


Fig. 5-8. Complete schematic diagram for the front-panel data switches, lamps, and plug connector.

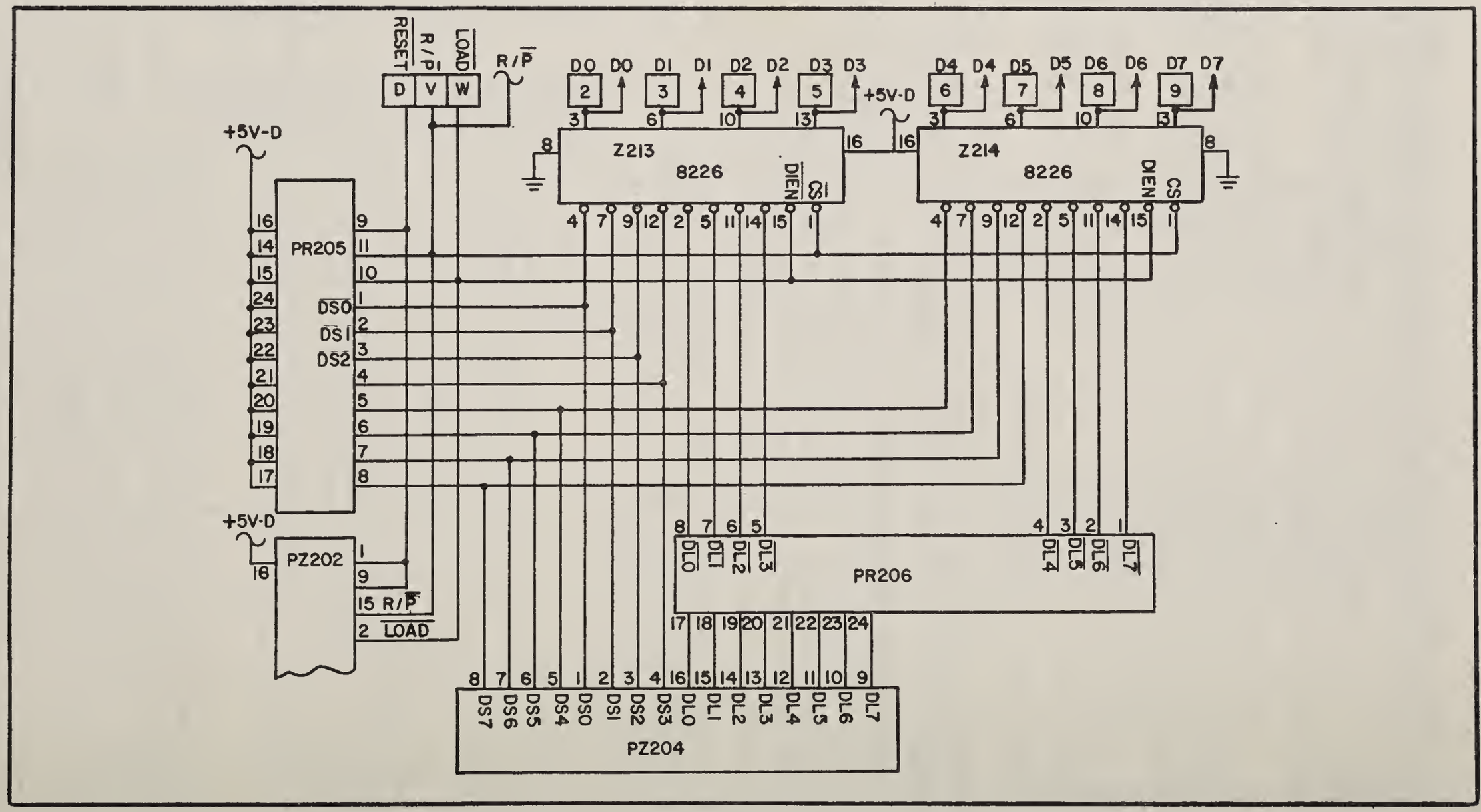


Fig. 5-9. Complete schematic diagram for the I/O board portion of the panel data switches, lamps, resistor network, and plug assemblies.

tion goes to the limiting resistors and, ultimately, the data indicator lamps on the front panel.

Pin 1 of Z213 determines whether or not the entire data switch assembly is operative. Note that this pin is connected to the $R/\overline{P}$ signal. Whenever this signal is at logic 0 (indicating the system is in its PROGRAM mode), Z213 is enabled so that there is an exchange of information between the data bus and data-switch circuitry. In the RUN mode, however, $R/\overline{P}$ is a logic-1 level, Z213 is completely disabled, and the data switch system is completely isolated from the data bus. Z214 serves exactly the same function and works the same way as Z213. The only difference is that Z214 handles the four higher-order data bits, D4 through D7.

When the front-panel system is accepting data from the data bus, all eight bits pass through 150-ohm limiting resistors on PR206 and to connections on plug connector PZ204. See the details for building PR206 in Fig. 5-10.

Returning again to the circuit in Fig. 5-8, it can be seen that the data lamp information from the plug-and-cable assembly goes to the cathodes of data indicator lamps DLO through DL7.

The three control switches on the front panel are handled in a similar, but much simpler fashion. After studying the circuit paths for the address and data schemes, you should have little trouble seeing how the same general principles apply to the three control switches.

Note in Fig. 5-9 that all three control signals are available at the 100-pin connector for the I/O board.

CONSTRUCTING AND TESTING THE FRONT PANEL

The mechanical features of the front-panel assembly are rather straightforward. A piece of 1/16-in. aluminum is cut so that it fits

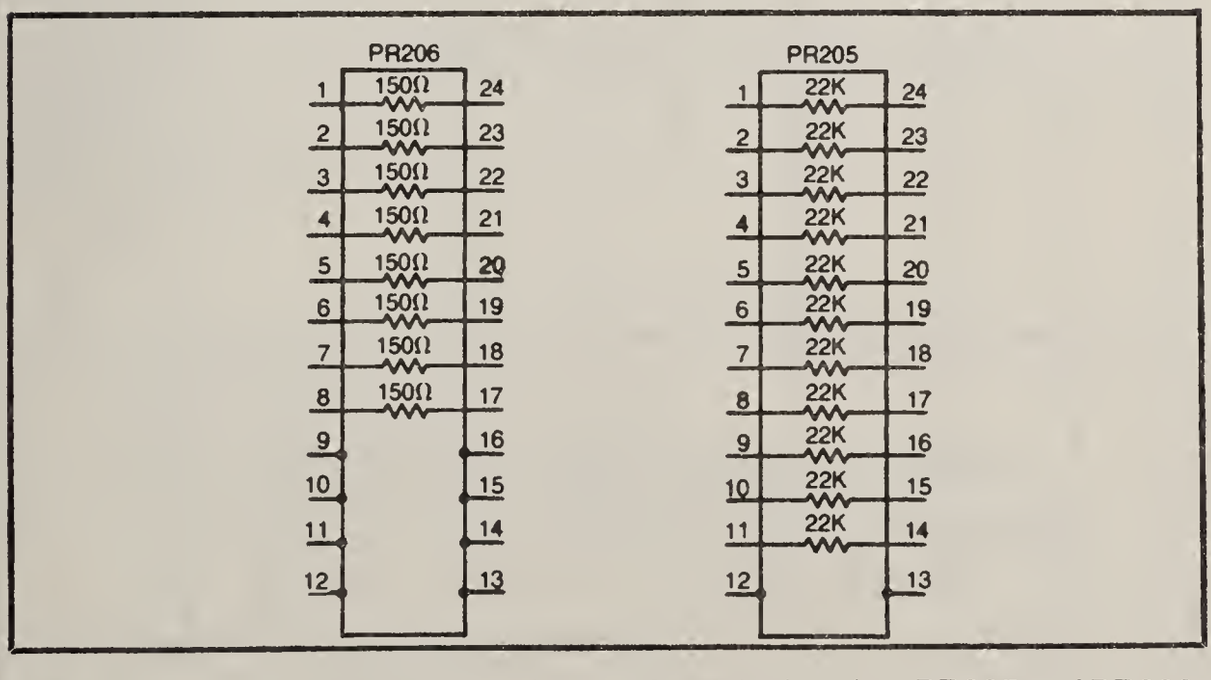


Fig. 5-10. Schematic and top-view layout of resistor headers PR205 and PR206.

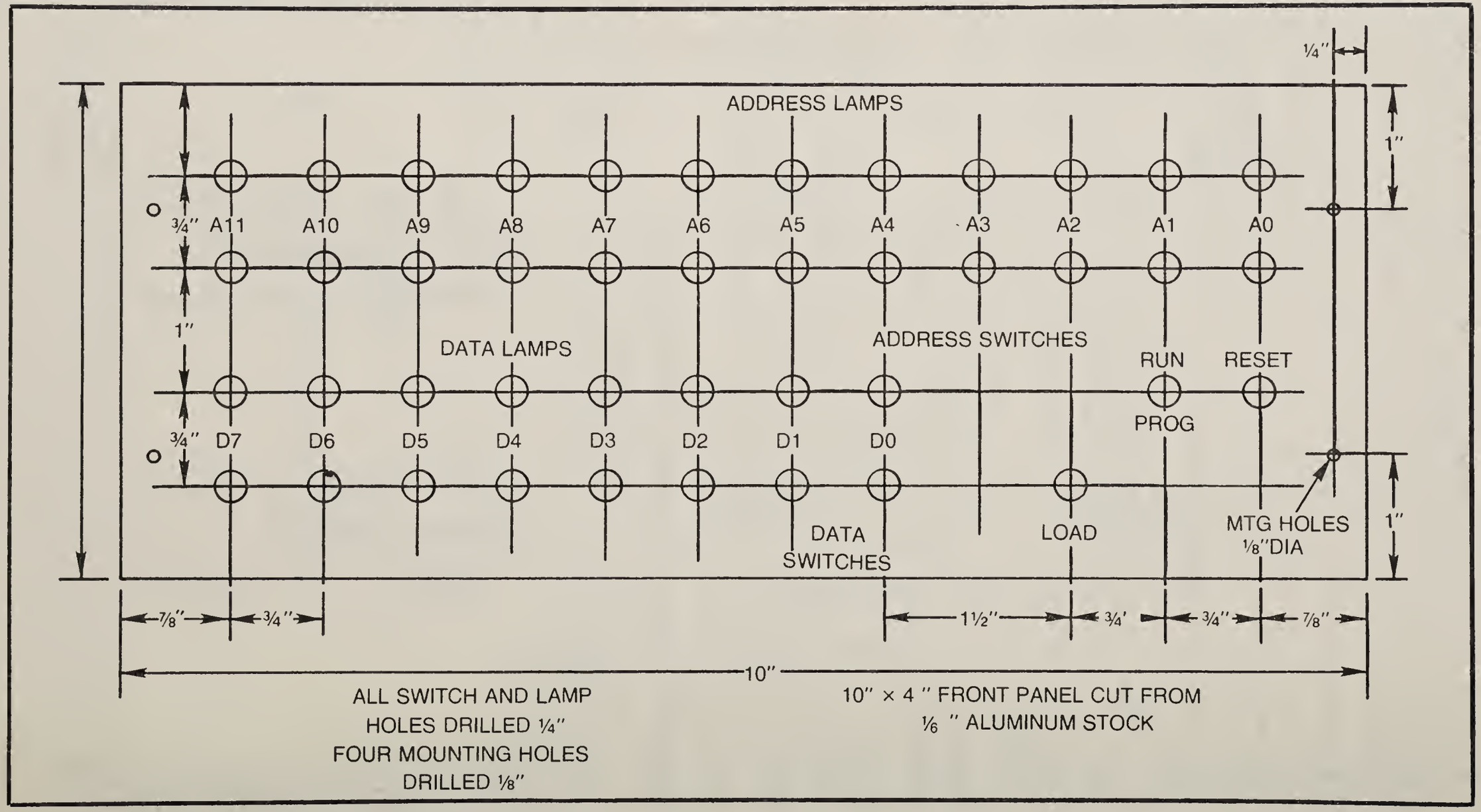


Fig. 5-11. Suggested mechanical layout of the front panel.

across the top of the card rack assembly. The other dimension has to be large enough to hold two rows of lamps and two rows of subminiature toggle switches.

A suggested layout and drilling plan for the front panel is shown in Fig. 5-11. If you use the parts specified in the front panel parts list (Table 5-1), all holes, except those for the four mounting screws, are ¼ in. in diameter.

After cutting and drilling the panel, you can give it a brushed-aluminum finish by scrubbing it lightly with crocus cloth. Maybe you'd rather paint it; this is the time to do it.

You can label the lamps and switches at this time, too. Dry-transfer decals look best; but, if you have neat handwriting, you can label them with the black ink of an ordinary-printed-circuit etch pen. In any event, it is a good idea to finish the panel with a coat or two of spray lacquer or clear enamel.

All the major parts for the front-panel assembly are listed in Table 5-1. Mount the parts in any order you like, but I found it easier to mount the switches first—the LEDs sometimes pop out of the Cliplite mountings before they are wired into place.

You must install the twelve address toggle switches with their OFF designations toward the *top* of the panel. This is necessary so that they will yield a logic-1 output whenever the handles are set to the upward position.

Since you will be using a hexadecimal number system for entering address information, you might want to use different colors for the switch handles. I used green covers on switches A0 through A3 (representing the lowest-order hexadecimal character), white covers on A4 through A7, and red handle covers on switches A8 through A11. None of this color coding is absolutely necessary, but it lends a snazzy appearance to the front panel and the codes will indeed prove useful later on.

Table 5-1. Front-panel parts list.

20 ea. Red LED (Jameco XC556R)
20 ea. Cliplite LED mounts (Jameco)
22 ea. SPST subminiature toggle switches (RS #275-324)
3 ea. 16-pin wire-wrap IC sockets (Jameco)
1 ea. Momentary pushbutton switch (Jameco FPF01/PB126)
3 ea. 16-conductor DIP jumpers (Jameco DJ16-1-16)

10-in. × 4-in. aluminum front panel
3-in. × 2-in. perfboard with 0.1-in. hole spacing

Kynar wire-wrap wire, assorted colors

Install the eight data switches with the OFF designations toward the *bottom* of the panel. Recall from the theory discussions that the data switches generate inverted or active-low signals. The eight data switches, in other words, must be mounted upside down relative to the twelve address switches.

Color code the handles of the data switches in groups of four. For my own system, I put red handle covers on switches D0 through D3 and white covers on D4 through D7. The data, you see, will also be entered in 4-bit hexadecimal notation, each color group representing a different hexadecimal character.

Install the RUN/PROGRAM and RESET switches with their OFF designations toward the bottom of the panel, the same way the eight data switches are installed. Color code the handles any way you like.

Install the LOAD pushbutton. It doesn't make any difference at this point whether one end is toward the top or bottom of the front panel. You will be compensating for the arrangement in either case later on.

Snap the twenty Cliplite holders into the lamp positions, and then install a red LED into each one of them. Don't worry about keeping track of which terminals are the anodes and which are the cathodes. You'll be working that all out in the next step.

Now is a good time to check the LEDs before wiring them permanently into place. Find a 1k resistor, attach a pair of alligator clip leads to the 12V DC output of your auxiliary power supply, and connect the 1k resistor to the free end of the positive lead. You are about to do two things: check each LED to determine whether it is good or bad, and determine which of the two leads is the anode.

Clip the negative lead from the auxiliary power supply to one lead on an LED, and then touch the free end of the 1k test resistor to the other lead of that same LED. If the LED lights up brightly, you know two things. First, you know the LED is a good one, and second, you know the anode is the lead you are touching with the resistor. Keep track of the anode lead by rotating the LED so that the anode is toward the top of the front panel.

If the LED does not light up when you test it in this way, reverse the connections. If it still doesn't light, it must be a dud (or else your power supply testing rig isn't working). But if the LED does light up brightly, you know the anode is the lead connected through the 1k test resistor to +12V.

Teach each one of the twenty address and data lamps this way. You probably won't find any bad LEDs, but the operation is neces-

sary for making doubly sure you know which leads are the anode leads.

Next, solder or wire-wrap all twenty LED anodes together. Note from Figs. 5-5 and 5-8 that the anodes of all the LEDs are connected together to the same +5V source.

Even after connecting all the anodes together, some of the LEDs might still pop out of their Cliplite mountings. I had that problem one time, so I glued each of the LEDs into the rear of the mountings with a dab of silicone rubber (an item commonly sold as "bathtub calk").

Wire together and solder the ON connections of all the toggle switches. This will eventually be the COMM bus shown for in Figs. 5-5 and 5-8.

Before wiring the LOAD pushbutton switch, however, you must determine which set of terminals represents the normally open configuration. Connect an ohmmeter between the center terminal on the LOAD switch and one of the other two terminals. If the ohmmeter shows zero ohms when you are *not* depressing the pushbutton, you have picked the wrong outside terminal. Try the center and the other terminal. Now the ohmmeter should show zero ohms only when the pushbutton *is* depressed. That outside terminal is the one you want to wire and solder to the COMM bus wiring to all the other switches.

The front-panel parts list in Table 5-1 specifies three 16-pin wire-wrap sockets and a small piece of perfboard. Arrange these three sockets in some neat fashion on the perfboard, and use a bit of silicon rubber or epoxy cement to hold them in place. These three sockets are PZ02, PZ03, and and PZ04 as shown in Figs. 5-5 and 5-8.

Strip and solder a length of Kynar wire to each of the OFF terminals of the twenty-two address, data, and control toggle switches. Wire wrap the opposite ends of these wires to the sockets as designated in Figs. 5-5 and 5-8. Connect the center terminal of the LOAD pushbutton to pin 2 of PZ02.

Be careful with this wiring operation. If you aren't 100 percent accurate, you are going to have problems later. Good solder connections at the switches are especially important. You will probably find yourself having more trouble with soldered connections than with the wire-wrapped ones.

Then wire wrap the cathode of each LED to its designated socket as shown in Figs. 5-5 and 5-8.

Finally, solder one end of a piece of wire-wrap wire to the COMM switch bus. Wrap the other end to pins 7 and 8 of PZ02. Wrap another length of wire from the anode of one of the LEDs to pin 16 to PZ202.

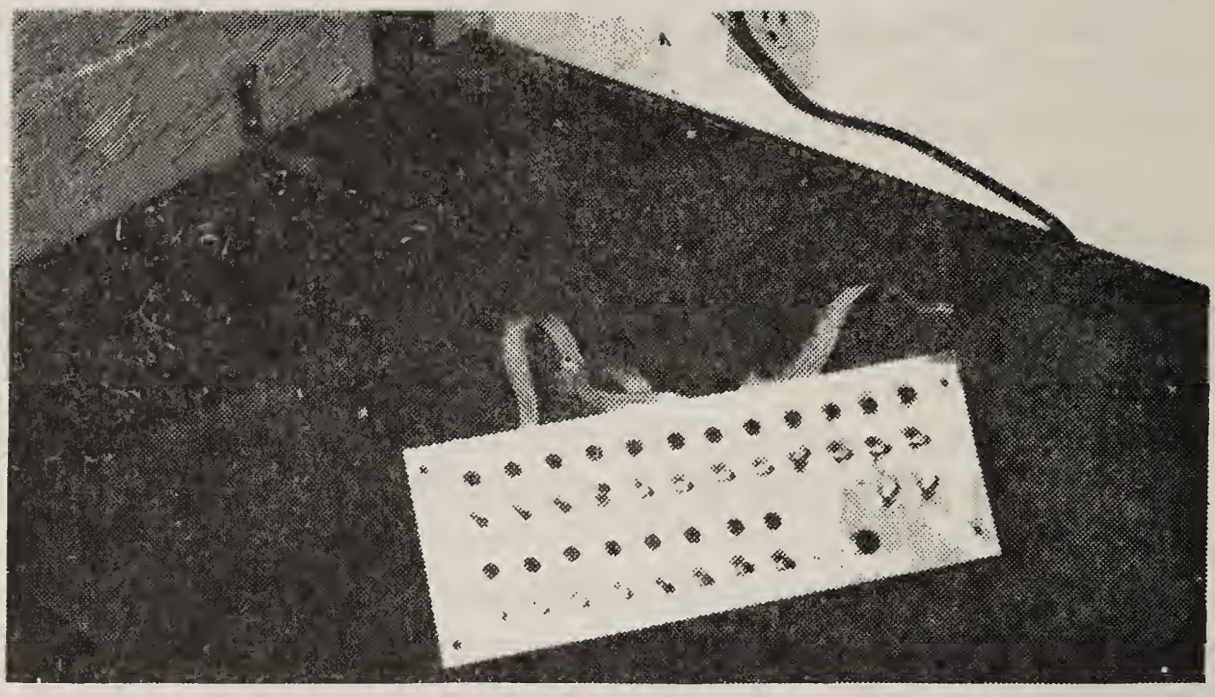


Fig. 5-12. Front panel assembly.

That completes the wiring operations for the front-panel assembly (Fig. 5-12). Neaten up the routing of all the fine wire and, if you want, bundle sections of the wiring together.

You can leave the plug card suspended from the wiring if you want. If the wiring is bundled tightly, you won't have any problem with things vibrating loose. That dangling piece of perfboard might offend your sensibilities; if that's the case, it can be fastened to the back of the front panel with a pair of long screws and 2-in. stand-offs.

Now that you have completed the easy part of the work in this chapter, it is time to turn attention to the I/O board.

WIRING THE DATA AND ADDRESS SECTION OF THE I/O BOARD

The I/O board is one of two large microcomputer boards used in the Rodney system. The 8801 board specified in Table 5-2 has a vast number of pre-tinned holes and a space in the upper left-hand corner provided for a +5V regulator. At least the space for the regulator appears in the upper left-hand corner if you are looking at the component side of the board.

This board is really intended for S-100 type computer mainframes, and I seriously doubt that the original designer had any thought of using it in a working robot, especially in a working robot that thinks for itself. So you are going to have to make a couple of minor modifications.

We need two separate +5V regulators on this board. Unfortunately, the +5V-D supply (the one used for the front-panel assembly) is not the one that goes in the neat little space allocated for a regulator. You must mount your own regulator in the lower right-hand corner.

This really isn't very hard at all. It simply means drilling a hole for the regulator mounting screw. To do this, arrange the TO-220 heat sink neatly in the lower right-hand corner of the board, and mark the place where the mounting screw will pass through the heat sink and board. Drill the special mounting hole at that point. Then mount the regulator (Z216) and the heat sink to the board. Slip a T46 wrap terminal into each of the three holes under the regulator's terminals and tack-solder the terminals to the board. Wrap the ends of the regulator terminals around their respective terminal pins and solder them into place. After connecting the INPUT terminal to the POWER strip running around the circuit board, and the COMM terminal of the regulator to the GND strip, you're all done.

Figure 5-13 shows how the other parts should be arranged on the I/O board. Arrange the sockets for the plugs, ICs, and resistor headers on the board. Bear in mind that you will be putting a lot of other parts on this same board later on, so there is little room for exercising freedom with regard to placing these components. Tack-solder each of the sockets at two or more places—say at pin 1 and the pin on the diagonal from pin 1.

Wire C217 into place between COMM and the +5V-D output of the regulator.

Slip the T46 pins for the card plug's data bus, address bus, and control connections. Figure 5-13 shows the relative positions of

Table 5-2. I/O-board parts list for front-panel operations.

ICs
Z209, Z210—74LS367 hex 3-state buffer (Jameco)
Z211, Z212—7407 hex inverter (Jameco)
Z213, Z214—8226 quad inverting bus transceiver (Jameco)
Z216—5V, 1A regulator, 7805 or LM340T-S (Jameco)

Resistors and Capacitors
C217—1000μF, 16 WVDC electrolytic (Jameco)
23 ea. 22k, ¼W resistor
20 ea. 150 Ohm, ¼W resistor

Wire-Wrap IC Sockets
2 ea. 14-pin WW socket (Jameco)
7 ea. 16-pin WW socket (Jameco)
4 ea. 24-pin DIP header (Jameco 24-pin HP)

Other Parts
1 ea. TO-220 heat sink (Jameco 291-.36H)
1 ea. Vector 8801, 100-pin circuit board (Jameco)
1 pk. Vector T46 WW terminal pins (Jameco)
Kynar WW wire, assorted colors
1 ea. 50/100 WW connector (Jameco R681-1)

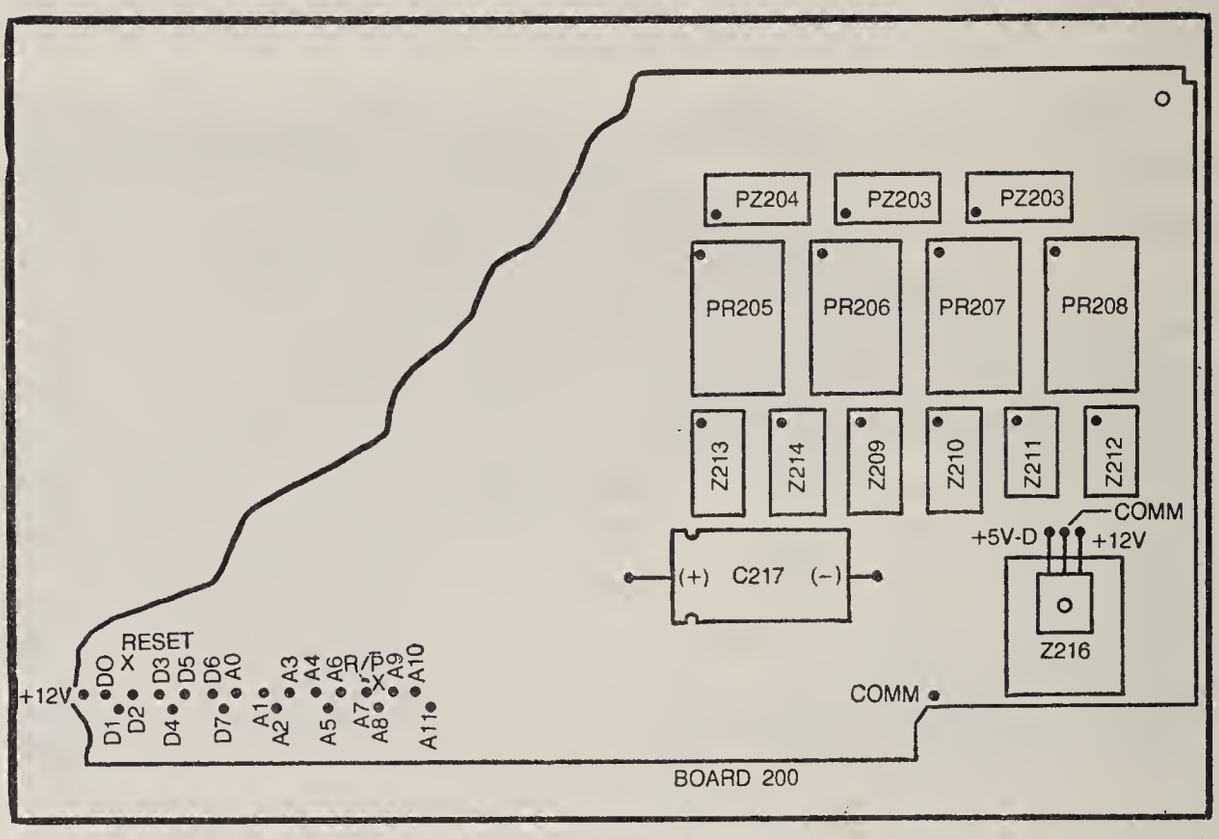


Fig. 5-13. Location diagram for I/O board components required for front-panel operations. More components will be added to this board at a later time.

these terminals, but Table 5-3 lists the positions in more exact detail. Wire-wrap the circuit as shown in Figs. 5-6 and 5-9.

Solder the 22k pull-up resistors and 150-ohm limiting resistors to the four 24-pin DIP headers as shown in Figs. 5-7 and 5-10. Again, let me warn you that solder connections are going to cause you some headaches later on if these connections aren't super-good ones. Every connection problem I had with my Rodney circuit was a solder connection.

If you haven't already attached the card connector for the I/O board to the card frame, you must do it now. See details in Chapter 4.

Place the resistor headers and ICs into their locations on the I/O board using the placement diagram in Fig. 5-13 as a guide.

Slip the I/O board into place and fasten the front-panel assembly to the top of the card rack. Finally, connect the three 16-conductor ribbon cables to their respective sockets on the front panel and I/O board. Figure 5-14 shows a schematic arrangement of these three cable assemblies.

FRONT-PANEL SYSTEM TESTS

After all the components are in place and the three ribbon cables have been installed, it is time to apply power to the front panel system. Connect the +12V output of the auxiliary power supply to pin 1 of the I/O board connector, and run the COMM connection to pin 100.

Turn on the power supply and watch what happens. If the thing starts to smoke or blows fuses, you're on your own. There is a wiring error—a short circuit—somewhere, and in this case it is up to you to check out the wiring.

What should happen at this point depends mainly on the setting of the control switches. Set the RUN/PROGRAM switch to PROGRAM. In this operating mode, all eight data lamps should go on, and they shouldn't respond at all to any changes in the setting of the data switches. If none of the indicator lamps go on, you probably have a defective connection between the output of the +5V-D regulator and the positive-bus connection to the anodes of all the LEDs. Check for +5V power on the LED anode box, pin 16 of PZ202 and PZ02.

In the event some of the data lamps go on, but others do not, chances are that the $\overline{DL}$ connections to the unlighted lamps are open somewhere along the line. Check the wiring all the way through, as well as solder connections to the corresponding 150-ohm limiting resistors on the I/O board.

Assuming now that all the data lamps go on when RUN/PROGRAM switch is in its PROGRAM position, check the data-

Table 5-3. I/O-board connector terminals required for front-panel tests and troubleshooting.

Component Side	Reverse Side
1—+12V	D(54)—$\overline{\text{RESET}}$
2—D0	W(69)—$\overline{\text{LOAD}}$
3—D1	V(68)—R/$\overline{\text{P}}$
4—D2	
5—D3	
6—D4	
7—D5	
8—D6	
9—D7	
10—A0	
11—A1	
12—A2	
13—A3	
14—A4	
15—A5	
16—A6	
17—A7	
18—A8	
19—A9	
20—A10	
21—A11	
50—COMM	
100-pin connections required for testing the front panel	

lamp circuitry in its entirety by grounding the data bus pins at the 100-pin connector, one at a time. Running a jumper between COMM and D0 (pin 2 on the main I/O socket connector) should make the D0 lamp go out. Likewise grounding the D1 pin should make lamp D1 go out, and so on through D7.

Any failure through this particular test indicates a bad wire-wrap or solder connection somewhere between the system data bus connections at the 100-pin connector and the data lamps. There are a lot of connections and components along this route, so you might want to use a logic probe, tracing the status of the signal one step at a time until you find the point where things are going wrong.

Assuming now that you have good communication between the data-bus connections at the 100-pin connector and the data lamps, depress the LOAD pushbutton. If the RUN/PROGRAM switch is still in its PROGRAM position, you should see the data lamps indicating the status of the eight corresponding data switches.

While depressing the LOAD button, switch the D0 data switch up and down. The D0 lamp should go on when the D0 data is up, and that same lamp should go out when the D0 switch is in its down position. Each of the eight data lamps should respond in this fashion to its corresponding data-switch setting, but only as long as the LOAD button is depressed and RUN/PROGRAM is in the PROGRAM position.

If none of these switch-and-lamp tests work, the problem is either due to a faulty connection at the switch ground bus or at the LOAD switch. The problem in this case might be at the bi-directional buffers (Z213 and Z214), but since these components have already been tested earlier, they aren't a likely cause of this problem. Any problem at this point can be traced down with a logic probe using the schematics in Figs. 5-8 and 5-9 as guides.

After running through all these data lamp and switch tests, and working out any problems, you can be reasonably certain that part of the panel circuitry is in good working order.

Testing the address lamps and switches is a somewhat simpler task. No matter what the setting of the RUN/PROGRAM switch might be, the address lamps should respond directly to the status of their corresponding address switches. Flipping the A0 switch up and down, for example, should cause lamp A0 to turn on and off accordingly. The same is true for all twelve address switches and lamps.

If none of the address lamps light up, regardless of the address-switch settings, the problem is most likely a defective power supply connection to either the lamps or the address switches. Check for +5V power on the common anode connections

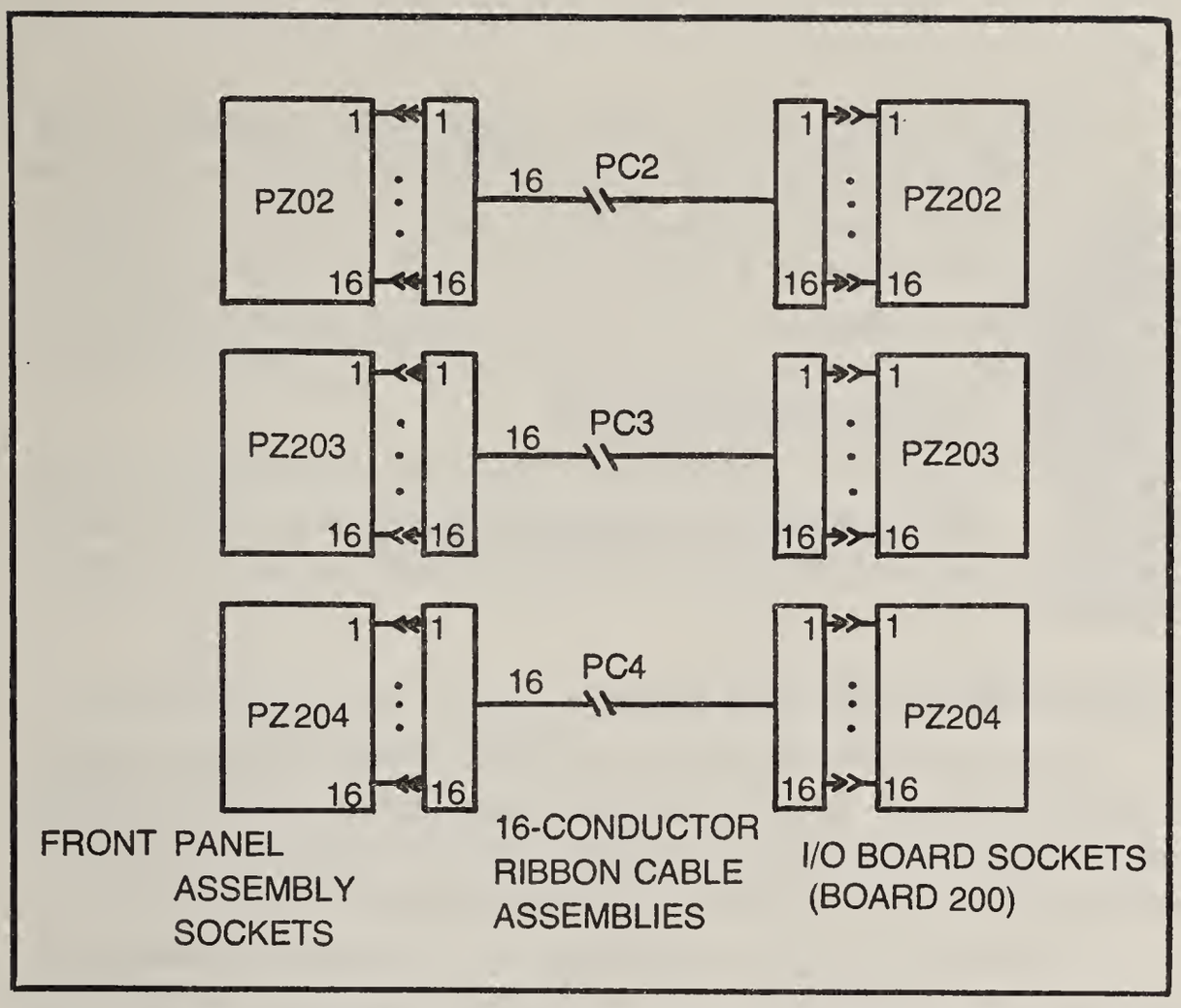


Fig. 5-14. Documentation of plug assemblies for the front panel and I/O board.

of the address lamps, and make certain the COMM bus wiring to the address switches is indeed connected to system ground.

The problem in this instance could also be in the cable connections between the front panel and I/O board, but it isn't likely that such a problem would affect all twelve address switches and lamps. The trouble might be in inverters Z211 and Z212 on the I/O board, but troubles at that place would more likely cause all the address lamps to go on, rather than off.

Also check out the power supply connections to PR207 and solder connections on PR208.

If you are having problems getting the address lamps to respond to the setting of their respective address switches, don't panic. It's probably a "simple" problem that only needs to be systematically tracked down.

After getting the address lamps responding to their switches, set the RUN/PROGRAM switch to PROGRAM and use a voltmeter or logic probe to see if address information is getting to the system address bus. To do this, toggle A0 switch between its "1" and "0" position, and observe the response at the A0 bus connection of the 100-pin socket (pin 10). The logic level at that bus connection should follow that of the A0 switch.

Check out each of the twelve address-bus terminals in this fashion. As long as the panel is in the PROGRAM mode, each of the bus connections should respond directly to the setting of its corresponding address switch. Failure of any or all of these little tests indicate either defective wiring to the address buffers (Z209 and Z210) or bad connections between those same buffers and the bus connections at the 100-pin socket.

The address-bus connections at the 100-pin connector of the I/O board will be undefined as long as the panel is set for its RUN mode. Depressing the LOAD pushbutton should not have any effect on the address circuit tests.

TESTING AND TROUBLESHOOTING

The process of assembling any part of this Rodney system involves making a large number of connections. Even the most careful builders are going to make mistakes now and then, and you will soon find those mistakes causing problems.

Troubleshooting thus really begins at the construction stage of the operation. The more care you exercise in building the circuit, the smaller the chances of having to deal with a problem later on.

But problems will arise, and they must be ironed out before going any further. Ignoring the problems won't make them go away, they will always come back to haunt you. Don't settle for the "close enough" philosophy. Every item must pass its tests with absolute certainty.

Dealing with a problem in the circuit is rarely easy. Actually the real problem is finding the cause of the trouble, and it turns out that troubles are hard to find by anyone using trial-and-error methods. So dealing effectively with trouble depends on understanding how the circuit in question is supposed to work. That's why we have gone into so much detail in the theory sections of this chapter.

If you find some part of a test failing to show clear and positive results, and you're getting frustrated trying to track down the trouble, you probably need an in-depth review of the theory of operation.

Knowing exactly how a circuit is supposed to work goes a long way toward helping you fix anything that might go wrong with it now or later. Ignoring the theory of operation or passing over it lightly will eventually cause a lot of headaches. Take the time and effort to understand the operation of the circuit thoroughly, and you will come out ahead in the long run.

6 Function- And I/O-Select Circuitry

As described in the overall system plan in Chapter 2, the microprocessor chip must be able to interact with three basic subsections: a program RAM, an optional ROM, and a host of I/O ports. The purpose of the function select circuitry described in this chapter is to sort out the sources and destinations of the microprocessor information.

There are another set of function-select operations, however, that were treated by a few words in Chapter 2. The case in point is the interaction of the front-panel system with the program RAM and I/O ports. In Chapter 5, you found that the front panel can, in a limited way, replace the operation of the microprocessor chip. This interaction between the front panel and program RAM will be critical for entering programs into the system later on. Plus, a similar kind of interaction between the front panel and I/O ports will prove to be invaluable for testing I/O circuitry long before the microprocessor chip is installed.

Presumably you have already built and checked out the front-panel system as described in Chapter 5. Now you will be assembling the vital function-select circuitry and testing it by means of front-panel PROGRAM operations. If the scheme works from the front panel, it will work from the microprocessor later on in the project.

THEORY OF OPERATION—AN OVERALL VIEW

Familiarize yourself with the glossary of function-select terms in Table 6-1. You might not be able to appreciate the full meaning of all these terms at this point in the game, but you should at least give it a try, and be prepared to refer to this information as you work your way through the chapter.

The diagram in Fig 6-1 is a simplified version of the main function select circuitry. The outputs of S1, S2, and S3 are, in a sense, connected together to the system address bus and control

Table 6-1. Glossary of terms for the function- and I/O-select circuitry.

$\overline{\text{CROM}}$—an active-low signal that selects optional ROM operations. Tested, but not used, in this chapter.
$\overline{\text{CRAM}}$—an active-low signal that selects program RAM operations.
$\overline{\text{CSIO}}$—an active-low signal that selects system I/O operations.
IOW0—an active-high signal that selects a writing operation to output port 0 (ACTL).
IOW1—an active-high signal that selects a writing operation to output port 1 (ACTH).
$\overline{\text{IOR0}}$—an active-low signal that selects a reading operation from port 0 (ENVL).
$\overline{\text{IOR1}}$—an active-low signal that selects a reading operation from port 1 (ENVH).
$\overline{\text{IOR2}}$—an active-low signal that selects a reading operation from port 2 (TSWR).
IOW4—an active-high signal that selects a writing operation to output port 4 (MAWL).
IOW5—an active-high signal that selects a writing operation to output port 5 (MAWH).
$\overline{\text{IOR4}}$—an active-low signal that selects a reading operation from port 4 (MARL).
$\overline{\text{IOR5}}$—an active-low signal that selects a reading operation from port 5 (MARH).
$\overline{\text{CSMM}}$—an active-low signal that selects main memory operations.
A8'A9', A10', A11'—address bits 8, 9, 10 and 11 directly from the microprocessor chip. These will not be used in this chapter.
SOD' —serial data output directly from the microprocessor chip. This will not be used in this chapter.
S1'—a status signal level from the microprocessor chip that signals either a up writing operation (S1' = 0) or a reading operation (S1' = 1). This will not be used in this chapter.
$\overline{\text{RD}}$' and $\overline{\text{WR}}$'—active-low READ and WRITE signals directly from the microprocessor. These will not be used in this chapter.
$\overline{\text{LOAD}}$—The user-determined data load signal from the front panel system described in Chapter 5.
R/P—The RUN/PROGRAM signal from the front panel system described in Chapter 5.
A8, A9, A10, A11—the four higher-order address bus signals. For the purposes of this chapter, they will be derived only from the corresponding front panel address switches.
R/$\overline{\text{W}}$—a system version of the READ/WRITE command. R/W = 0 whenever data is to be written onto the data bus, and R/W = 1 when data is to be taken from the data bus.
RD—a buffered version of RD'
SOD—a buffered version of SOD'. Not used in this chapter.

bus. The information on these two busses comes from one of two places: the microprocessor chip via S1, or the front panel system via S2 and S3.

It is the R/$\overline{\text{P}}$ status that determines whether this information is supplied by the microprocessor or front-panel system. Whenever R/$\overline{\text{P}}$ is a logic 1, the system is in the RUN mode and accepts the bus information from the microprocessor through buffer S1. In the PROGRAM mode, however, R/$\overline{\text{P}}$ is a logic 0 and buffers S2 and S3 are enabled so that front panel information is placed onto the busses. All of this should sound rather familiar in the context of the discussions in Chapter 5.

Incidentally, you have already installed and tested the block in Fig. 6-1 that is labeled S3. This is the set of address buffers desig-

nated Z209 and Z210 in Fig. 5-6. Address bus connections A8 through A11 are the only ones shown in Fig. 6-1 because they are the only ones relevant to the function-select processes.

Before looking at the operation of the S4 block in Fig. 6-1, take a more detailed look at the role R/$\overline{P}$ plays in all of this. Whenever the system is in the normal RUN mode, R/$\overline{P}$ is at logic 1. Buffers S2 and S3 are disabled so that none of the front-panel signals pass to the address and control buses. At the same time, though, buffer S1 is enabled, and information from the microprocessor chip (if it is installed) goes to those same bus connections.

For instance, in the RUN mode, A8′ from the microprocessor goes to address bus line A8, A9′ goes to bus line A9 and so on. Signal S1′ from the microprocessor also passes to the control bus, but this particular signal is then renamed R/$\overline{W}$ to give it more meaning during construction and troubleshooting operations.

Along the same line of thinking, $\overline{RD}$′ and SOD′ become known as $\overline{RD}$ and SOD after passing through buffer S1 in the RUN mode. It appears, however, that the $\overline{WR}$′ signal from the microprocessor is lost somewhere along the line. This is not really the case, as you will discover later in this chapter. Remember, Fig. 6-1 is a *simplified* block diagram, and when considering simplified diagrams, it isn't always possible to tie in all the loose ends.

Placing the system into the PROGRAM mode isolates all the microprocessor signals from the system and substitutes those from the front-panel system. Information from A8 through A11 on the front panel becomes A8 through A11 on the address bus, and LOAD is renamed R/$\overline{W}$ as it passes through S2 to the control bus.

In summary, S1, S2, and S3 are function-select circuits. When the system is set up for its RUN mode, S1 delivers relevant microprocessor information to the four high-order bits of the address bus

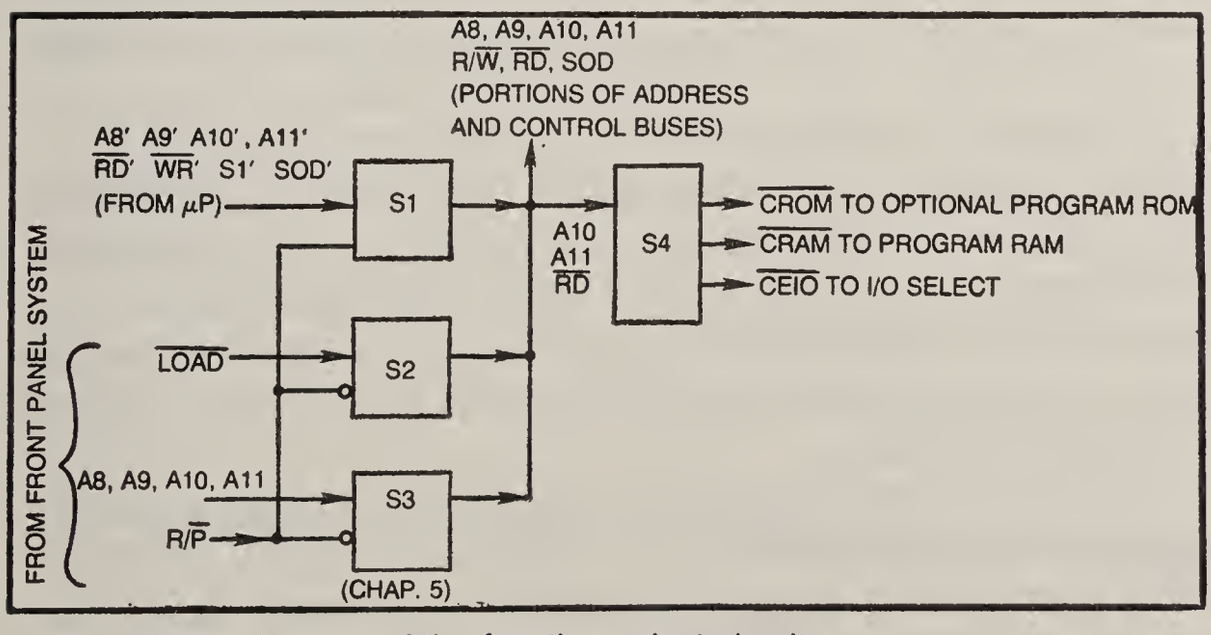


Fig. 6-1. Block diagram of the function-select circuitry.

and three of the control-bus lines. In the PROGRAM mode, that same set of bus lines gets its information from the front panel system.

Now, take a look at select circuit S4 in Fig. 6-1. The select operations here are somewhat different from those used for the three preceding circuits. The main job of S4 is to determine which one of the three basic sources or destinations of information will be enabled: the optional program ROM ($\overline{\text{CROM}}$ signal), the program RAM ($\overline{\text{CRAM}}$ signal), or any of the I/O ports ($\overline{\text{CEIO}}$ signal).

The Rodney computer system uses something called memory-mapped I/O. This is relevant to the discussion at hand, because it means one of the three major sources and destinations of information is selected (addressed) by the two higher-order address lines, A10 and A11.

It turns out that RAM operations are selected whenever A10 and A11 are both at logic 0 at the same time. In other words, the $\overline{\text{CRAM}}$ output of block S4 goes to logic 0 whenever A10 = A11 = 0. By the same token, ROM operations are elected whenever A10 =1 and A11 =0, and finally, one of the I/O port operations will be selected only when A10=0 and A11=1.

You will find out exactly how S4 does this job in the next section which deals with the finer details. For the time being, it is sufficient to realize what S4 does in a basic sense. You should note in passing, however, that S4 is enabled and fully operational whether the system is in the RUN or PROGRAM mode. In fact, all of the work you have done on the front-panel system pretty much ends right here.

The second portion of the system under consideration here is the I/O-select circuitry. Once you select an I/O operation by means of the select logic in block S4 of Fig. 6-1, there is still a need to sort out the individual I/O from among the six possibilities. This is the job of the I/O select circuit blocked out in Fig. 6-2.

Selector block S5 in Fig. 6-2 actually serves six I/O ports. Some of these ports are capable of both reading and writing information, while one is simply a read-only port. We will not be constructing the I/O ports, themselves, in this chapter, but much of the work here involves building and testing the circuitry necessary for selecting the I/O ports and determining whether they will accept or put out data bus information.

The system includes four general-purpose, 8-bit I/O ports that are selected by IOW0, $\overline{\text{IOR0}}$, IOW1, $\overline{\text{IOR1}}$, IOW4, $\overline{\text{IOR4}}$, IOW5 and $\overline{\text{IOR5}}$. The signals having the same suffix—0, 1, 4, or 5—enable the same ports. IOW0 and $\overline{\text{IOR0}}$, for instance, both service I/O-port 0, while IOW1 and $\overline{\text{IOR1}}$ both service port 1, and so on.

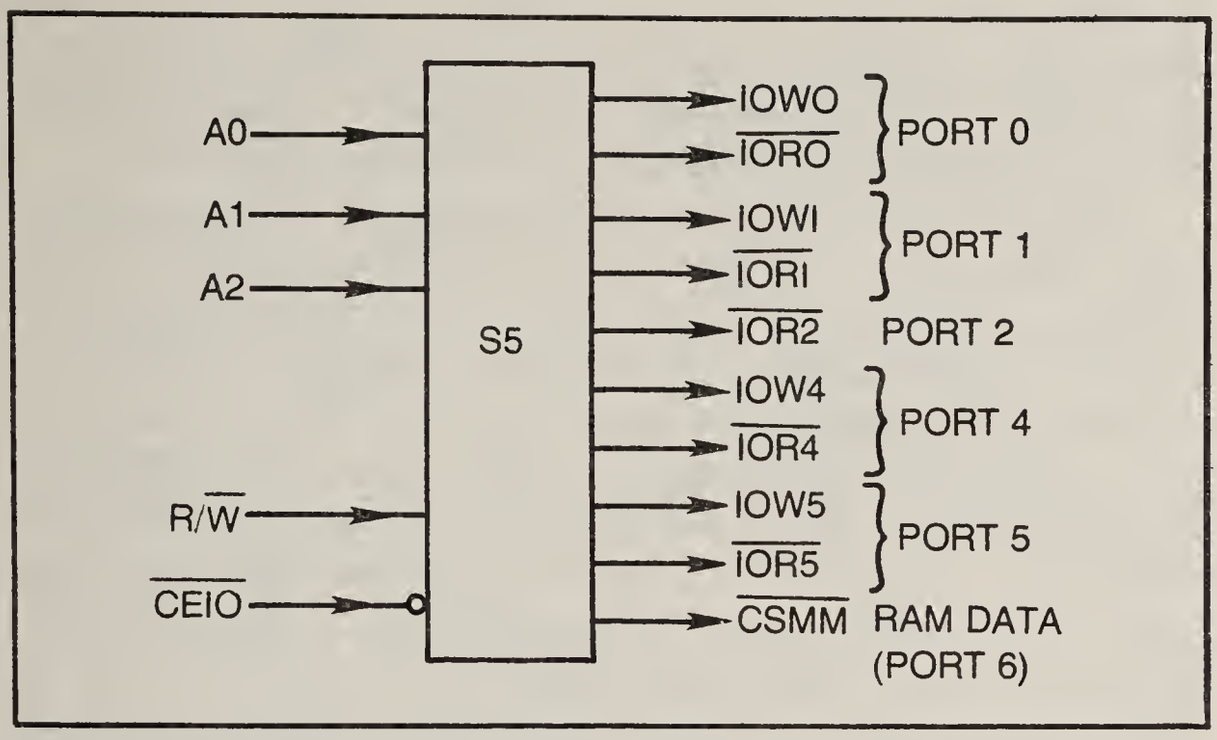


Fig. 6-2. Block diagram of the I/O-select circuitry.

The only difference between the two signals going to one of the I/O ports is that the "W" versions cause 8-bit bus data to output from that port, and the "R" versions of the enabling signals cause data-bus information to be entered via that port. Port 2 in the Rodney system happens to be a read-only port, so it has only an "R" designation, $\overline{\text{IOR2}}$. In passing, note that the select signals for "W" or writing operations are active high, while those for "R" or reading operations are active low.

Signal CSMM from S5 in Fig. 6-2 is a special sort of enabling signal for for the main-memory system. This signal could be labeled as port 6. It enables the memory circuit whether the system calls for a read or a write. It is thus both a read and write port, as most of the others are, but it makes no distinction between the two kinds of operations. If it is helpful to you, you might view CSMM as the result of the logical oepration $\overline{\text{I0W6} + \text{I0R6}}$.

Let me repeat that the outputs from S5 are not the ports, themselves, but rather the enabling signals for the I/O ports. You will not begin working with the actual port circuitry until Chapter 7.

The I/O-select block, S5, is enabled only when it receives a logic-0 level at its CEIO connection. This signal comes from block S4 in Fig. 6-1. Address bus lines A0, A1, and A2 then determine which one of the I/O functions will be selected, and the R/$\overline{\text{W}}$ input determines whether the circuit will select a read or a write operation.

There is little point in dwelling on the logic operations involved in this selection process until you have a chance to see the actual circuit. It is enough at this point to know that the I/O select operations calls for the five inputs just mentioned.

DETAILED THEORY OF OPERATION

The circuit shown in Fig. 6-3 details the exact wiring of the function-select circuitry described in connection with the block diagram in Fig. 6-1. It takes little more than a glance to identify the input and output signals already described and listed in Table 6-1.

Z103 and Z104 are both hex 3-state buffers with two active-low enable connections. Note that pin 15 on Z103 and Z104 are connected together at the $R/\overline{P}$ signal source. This source is the same RUN/PROGRAM signal developed in the front-panel system.

The other enabling pins (pins 1) are similarly connected together, but in this case they are connected to an inverted version of the basic $R/\overline{P}$ signal. This signal, after being inverted by Z110-A, can legitimately be called $R/\overline{P}$. Connecting the enabling pins in this configuration enables certain buffers within the IC when $R/\overline{P}=1$, disabling the same buffers when $R/\overline{P}=0$. Suppose, for the sake of this discussion, that $R/\overline{P}$ is logic 0. The system, in other words, is operating in its PROGRAM mode. In this case, the buffer controlled by enabling pin 15 will be disabled, while those controlled by pin 1 will be enabled. It happens that pin 15 on both of these buffer circuits controls information from pins 12 and 14, while pin 1 controls signals present on pins 2, 4, 6, and 10. So when $R/\overline{P}$ is set to logic 0, as in this example, the signals going to input pins 12 and 14 on both buffer ICs will pass through to output pins 13 and 11 respectively.

Since the $R/\overline{P}$ signal is inverted by Z110-A before it is applied to pin 1, it follows that the inputs controlled by these pins will be blocked. In other words, the data applied to pins 2, 4, 6, and 10 of both buffer circuits will be effectively gated off.

What does all this mean in the context of system RUN and PROGRAM operations? It means that information from the microprocessor (A8′ through A11′, SOD′, S1′, etc.) will be blocked at the buffers. The microprocessor signals are isolated from the rest of the system as long as the PROGRAM mode is in effect.

While the microprocessor information is being blocked by Z103 and Z104 in the PROGRAM mode, some signals are passing through the enabled portions of the two buffers. Pin 14 of Z103, for instance, passes the $\overline{\text{LOAD}}$ signal level through to pin 13, while at the same time, output pin 11 is held at logic 0 because its corresponding input pin, pin 14 of Z103, is permanently grounded.

All of this changes whenever the system is put into its RUN mode. Microprocessor information is allowed to pass freely through the buffers, with the $\overline{\text{LOAD}}$ signal and fixed levels inhibited.

If you'd like a more specific example of how this scheme works, consider the $R/\overline{W}$ output of the circuit in Fig. 6-1. This connection

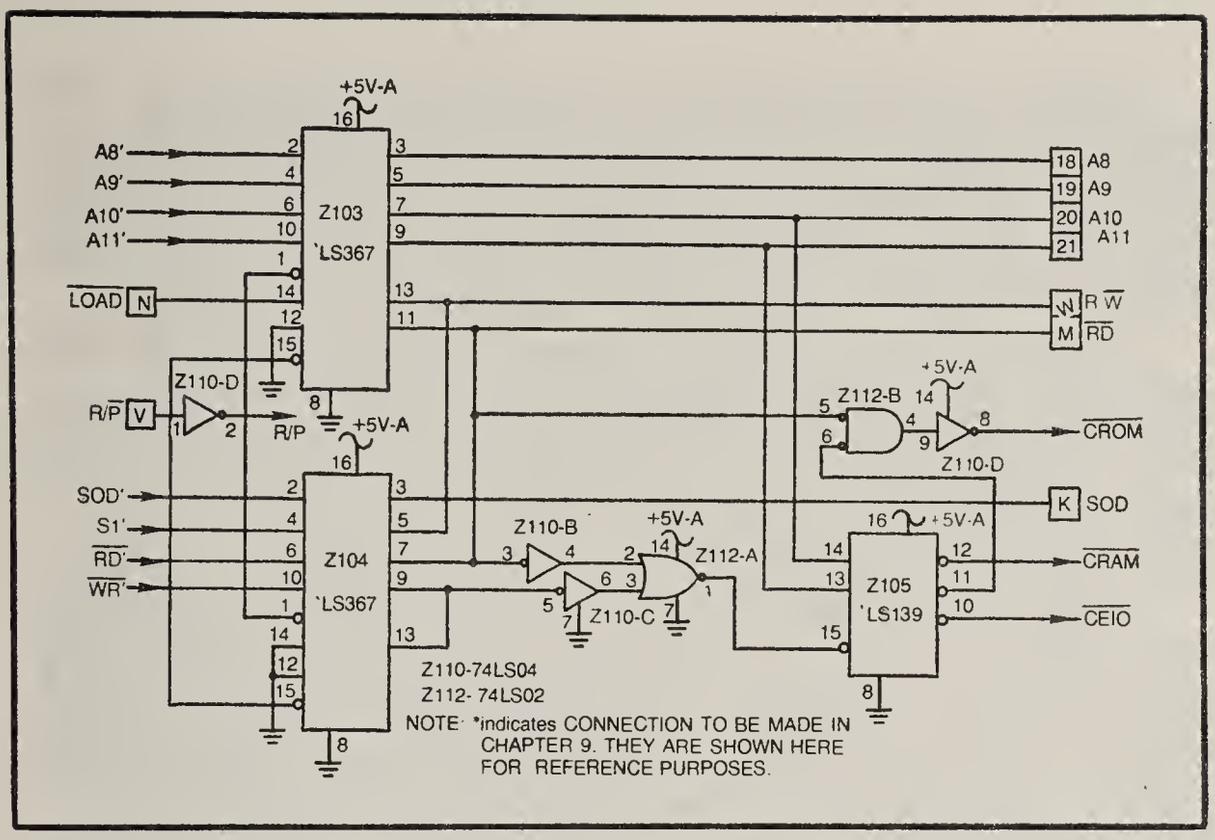


Fig. 6-3. Schematic diagram of the function-select circuit.

can be traced back to two different sources of signals, pin 13 of Z103 and pin 5 of Z104. Since a single point cannot tolerate logic signals from more than one source at a time, there has to be some provision for making certain only one of these two sources is providing data at any given moment. Because of the arrangement of the inverter, Z110-A, and the enabling pins on the buffer ICs, you should be able to see that pin 13 of Z103 and pin 5 of Z104 can never be enabled and providing information at the same time. One or the other is at work, but never both.

Therefore, we can tie these two sources of data together at the R/$\overline{W}$ output terminal without ever running into a conflict of information. More specifically, setting the system up for PROGRAM operation enables pin 13 of Z103, and the R/$\overline{W}$ output is effectively connected to the $\overline{LOAD}$ signal. Putting the system into the RUN mode, on the other hand, connects R/$\overline{W}$ to the S1′ output of the microprocessor. The R/$\overline{W}$ signal is thus either $\overline{LOAD}$ or S1′, depending on whether the system is set up for PROGRAM or RUN.

The same general idea applies to any pair of buffer outputs that are connected together in Fig. 6-3. In fact, it also applies to the four address bus lines, A8 through A11. In this latter case, however, the second source of information is not shown. It is really the identically labeled address outputs of Z201 on the I/O board. (See Fig. 5-6). In the PROGRAM mode, bus signals A8 through A11 come from the front panel circuitry, but in the RUN mode, they come from the microprocessor and Z103.

Z105 in Fig. 6-3 is the system's source of enabling signals. This device is actually a 2-line to 4-line decoder/demultiplexer. Its four active-low outputs are normally held at logic 1 and drop down to logic 0 only when (1) the circuit is enabled by a logic 0 at pin 15 and (2) one of the four outputs is addressed by signals at pins 13 and 14.

Suppose Z105 is enabled by having a logic-0 level at pin 15. This isn't always the case, but it will suffice for this part of the discussion. Further, suppose pins 14 and 13 of that same IC are at logic 0. Under this particular set of circumstances, output pin 12 will be at logic 0, and the rest of the outputs will show their normal, logic-1 output.

If the signals at pins 13 and 14 are then changed, but enabling pin 15 remains at logic 0, one of the other three outputs will go low and pin 12 will return to logic 1. If pin 14 is a logic 1 and 13 is a logic 0, the pin 11 output is the one—the only one— that is a logic-0 level. The pin-10 output goes low whenever pin 14=0 and pin 13=1.

All of this begins to mean something if you trace these pins to their source and destination. Note, for instance, that the addressing pins (pins 13 and 14) are connected to address bus lines A10 and A11. Further note that the three outputs are related to select signals $\overline{\text{CRAM}}$, $\overline{\text{CROM}}$, and $\overline{\text{CEIO}}$. The three main select signals, in other words, are themselves selected by combinations of 1s and 0s at A10 and A11.

Still assuming that pin 15 of Z105 is fixed at logic 0, you should now see that $\overline{\text{CRAM}}$ is energized whenever A10=A11=0—whether the system is in the RUN or PROGRAM mode. $\overline{\text{CEIO}}$ is likewise generated when A11=1 and A10=0. This output also operates the same way in either the RUN or PROGRAM mode.

$\overline{\text{CROM}}$ is different, however. Its behavior depends on the RUN/PROGRAM status. That's why it is taken from a circuit that works, at least in part, from the Z105 decoder. Z112-B and Z110-D work together as a 2-input NAND gate that has active-low inputs. The $\overline{\text{CROM}}$ output of Z110-D goes low only when the two inputs to the Z112-B device are both low.

Tracing the signals back, the pin-6 input of Z112-B goes low whenever Z105 is enabled and A10=1 and A11=0. The pin-5 input to Z112-B depends on the RUN/PROGRAM status of the system. If the system happens to be in the RUN mode, pin 5 gets its information from pin 7 of Z104 which can eventually be traced back to the $\overline{\text{RD}}'$ terminal of the microprocessor. In the RUN mode we can get an active-low, zero-energizing level from $\overline{\text{CROM}}$ only when that device is selected by A10 and A11 and the microprocessor is generating a $\overline{\text{RD}}'$ signal. Go to the PROGRAM mode, though, and pin 5 of Z112-B is always fixed at logic 0, because its corresponding buffer input (pin 12 of Z103) is connected to ground.

All of this part of the discussion about Z105 deals only with its addressing feature. As of now, there has been little of any significance related to the all-important enabling pin, pin 15. We've just been assuming it is always at logic 0 so the addressing/selection process can take place.

However, what if pin 15 of Z105 is a logic 1 instead? In this case, all three of the select outputs go to logic 1 and stay there. There is no selection process, no matter what A10 and A11 are doing.

Pin 15 of Z105 is fed from a second kind of logic circuit, one that works as a NOR gate having active-low inputs. The inputs in this case are from pins 3 and 5 of Z110; pin 15 of Z105 will be pulled down to logic 0 if either pin 3 OR pin 5 of Z110 goes to logic 0. Looking at the situation the other way around, pin 15 will be held at logic 1 (thereby disabling any decoding action) as long as pins 3 and 5 of Z110 are a logic 1. From this perspective, the little logic circuit works as a 2-input AND gate.

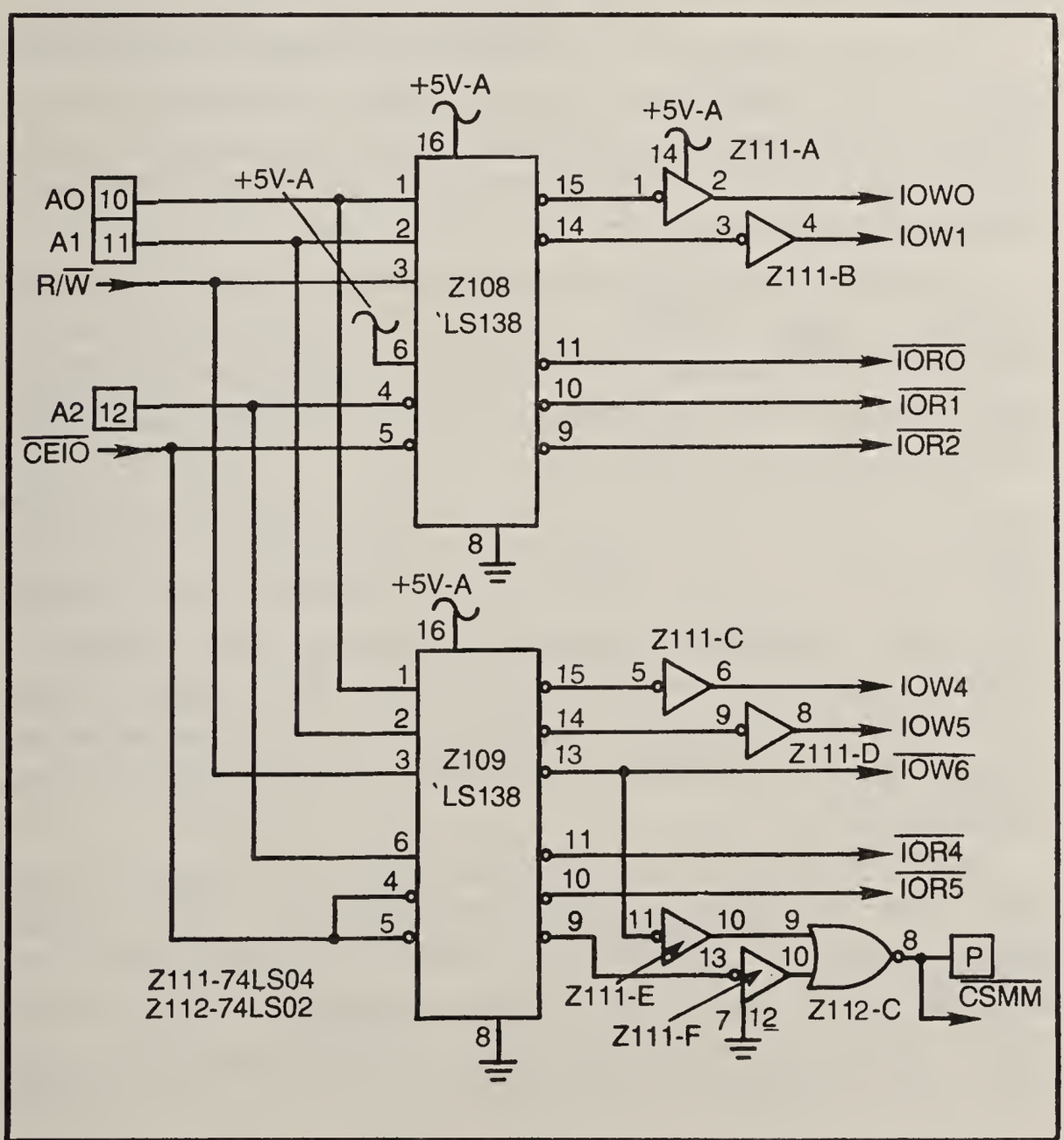


Fig. 6-4. Schematic diagram of the I/O-select circuit.

The important thing to do now is figure out what the inputs to Z110 are supposed to be. When the system is in its RUN mode, pins 3 and 5 of Z110 are connected via Z104 to the $\overline{WR}'$ and $\overline{RD}'$ signals from the microprocessor. In the RUN mode, then, pin 15 of Z105 enables the selection process when the microprocessor calls for a read or write operation. Otherwise the select circuitry is disabled in the RUN mode.

Whenever the system is set up for PROGRAM operations, pins 3 and 5 of Z110 are logic 0 due to the buffer connectoins at Z103 and Z104. The select circuit, Z105, is always enabled during PROGRAM operation—a feature that is especially valuable for checking out all the memories and I/O ports from the front-panel system.

The circuit in Fig. 6-4 shows the I/O-select circuitry that was described in a general fashion in the previous section of this chapter. See Fig. 6-2 for the general block diagram of this circuit.

Z108 and Z109 in Fig. 6-4 are both 3-line to 8-line decoders. They are enabled only when pin 6 is a logic 1 and pins 4 and 5 are logic 0. Whenever these devices are enabled, the signals at pins 1, 2, and 3 determine which of the eight different outputs will go to logic 0.

The precise details of this circuit are perhaps best described in terms of a truth table. See Table 6-2. This truth table is not only helpful for explaining the operation of the I/O-select circuit, but it can prove invaluable if it becomes necessary to test and troubleshoot the select system later on.

According to the first line in the truth table, the entire circuit is disabled as long as $\overline{CEIO}$ is a logic 1. This should make sense because the $\overline{CEIO}$ signal is the active-low enabling signal. If you aren't selecting an I/O operation of one kind or another, all of these outputs should be disabled.

Now suppose you enable the I/O-select circuitry by letting $\overline{CEIO}$ go to logic 0. What happens after that depends on the status of $R/\overline{W}$, A2, A1, and A0. If $R/\overline{W}$ is set to 0 as shown in the second column of the truth table, you begin carrying out a series of possible "writing" operations based on the three binary address inputs. If A2, A1, and A0 happen to be at logic 0, output IOW0 is enabled as indicated on the first line of the truth table. Setting A0 to logic 1 then enables IOW1.

About half-way through the truth table, $R/\overline{W}$ changes to its logic-1 "read" condition, and the three address lines enable reading operations such as $\overline{IOR0}$, $\overline{IOR1}$, etc. With the notable exception of the $\overline{CSMM}$ output, each of the outputs is enabled by only one set of input conditions. As mentioned earlier, $\overline{CSMM}$ is enabled at the same address designation, regardless of the status of the $R/\overline{W}$

Table 6-2. Truth table for the I/O-select operations.

$\overline{CEIO}$	R/$\overline{W}$	A2	A1	A0	IOW0	IOW1	IOW4	IOW5	$\overline{IOR0}$	$\overline{IOR1}$	$\overline{IOR2}$	$\overline{IOR4}$	$\overline{IOR5}$	CSMM
1	X	X	X	X	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	1*	0	0	0	1	1	1	1	1	1
		0	0	1	0	1*	0	0	1	1	1	1	1	1
		0	1	0	0	0	0	0	1	1	1	1	1	1
		0	1	1	0	0	0	0	1	1	1	1	1	1
		1	0	0	0	0	1*	0	1	1	1	1	1	1
		1	0	1	0	0	0	1*	1	1	1	1	1	1
		1	1	0	0	0	0	0	1	1	1	1	1	0*
		1	1	1	0	0	0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0	0	0*	1	1	1	1	1
		0	0	1	0	0	0	0	1	0*	1	1	1	1
		0	1	0	0	0	0	0	1	1	0*	1	1	1
		0	1	1	0	0	0	0	1	1	1	1	1	1
		1	0	0	0	0	0	0	1	1	1	0*	1	1
		1	0	1	0	0	0	0	1	1	1	1	0*	1
		1	1	0	0	0	0	0	1	1	1	1	1	0*
		1	1	1	0	0	0	0	1	1	1	1	1	1

Asterisks (*) indicate the active states

signal. The unique behavior of the CSMM output is determined by Z112-C, Z111-E, and Z111-F in Fig. 6-4.

ASSEMBLING THE SELECT CIRCUITRY

All of the function- and I/O-select circuitry described so far in this chapter goes onto the second main circuit board. This board will be known as the CPU Board, or Board 100. It takes these names because it will ultimately hold the microprocessor chip and all the parts on this board have a 100-series prefix. By the time you complete the work in this chapter, you will be working with partly assembled versions of the two main circuit boards. This means you will also have two 100-pin connectors attached to the system's card rack assembly, and there will be some wire-wrap connections between the connectors.

Setting Up the CPU Board

Figure 6-5 shows the arrangement of components on the left-hand side of the CPU circuit board. As part of the work in this chapter, you will be installing the sockets and components that carry part-number designations. The other components, shown only in outline form, will be installed at a later time and are shown here only to help you locate the positions for the parts relevant to this chapter.

You might have noted from the schematics in the previous section that the ICs are operated from a +5V-A power source. This is one of two 5V, regulated supplies that will fit onto the CPU board. For the time being, however, you are only interested in the +5V-A supply shown in the upper left-hand corner of Fig. 6-5.

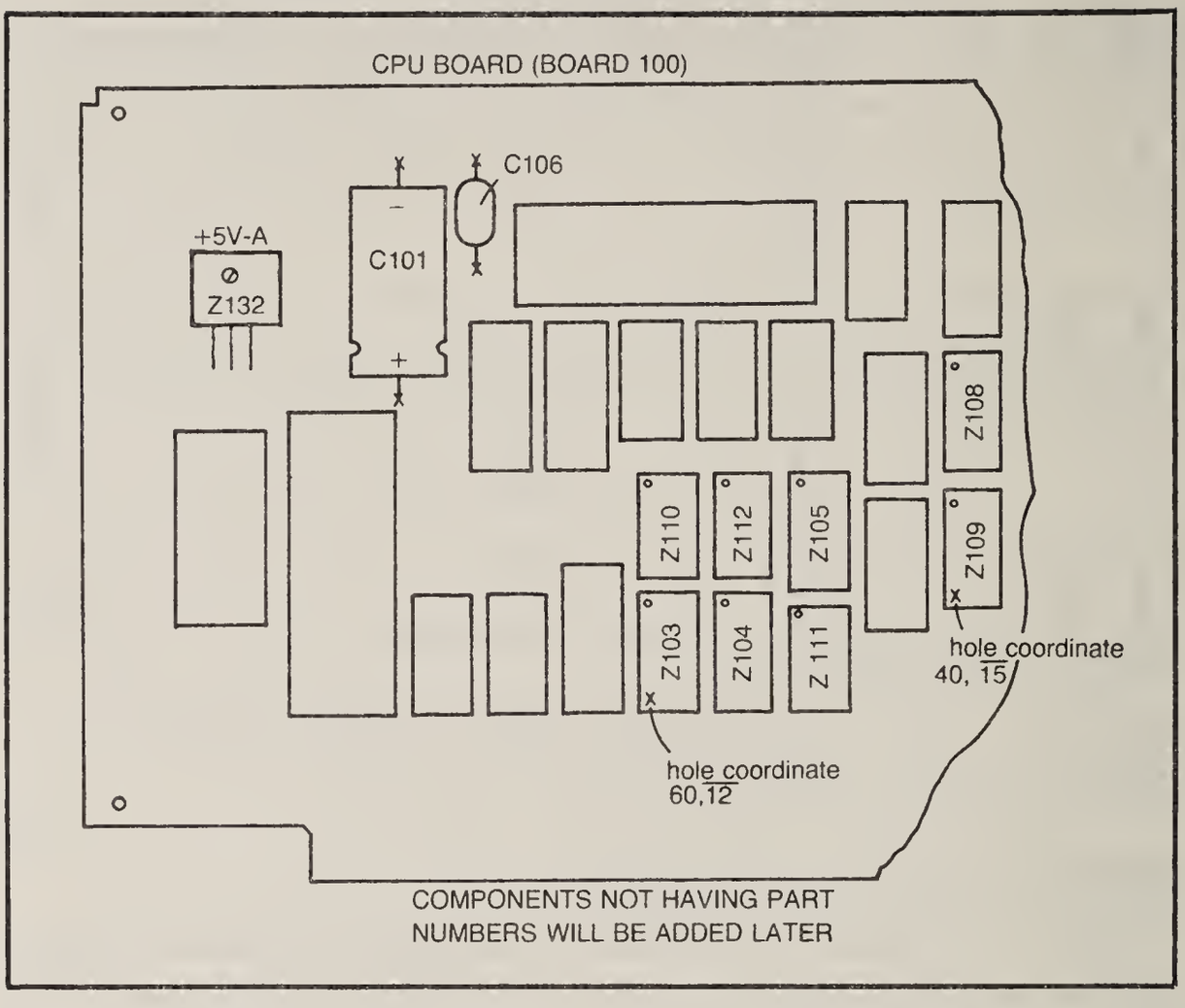


Fig. 6-5. Physical layout of function- and I/O-select circuitry on the CPU board (Board 100).

Attach the regulator, Z132, in the designated area of the CPU board, using one of the TO-220 heat sinks. Also install the filter and deglitching capacitors, C101 and C106, using the power supply circuit in Fig. 6-6 as a wiring guide.

As an aid for locating the position of the ICs to be installed at this time, note that the 8801 circuit board has sets of numerals arranged around the outer edges. These designate the horizontal and vertical coordinates of each hole on the board. The lower left-hand pin on Z103, for instance, should go into a hole at horizontal coordinate 60

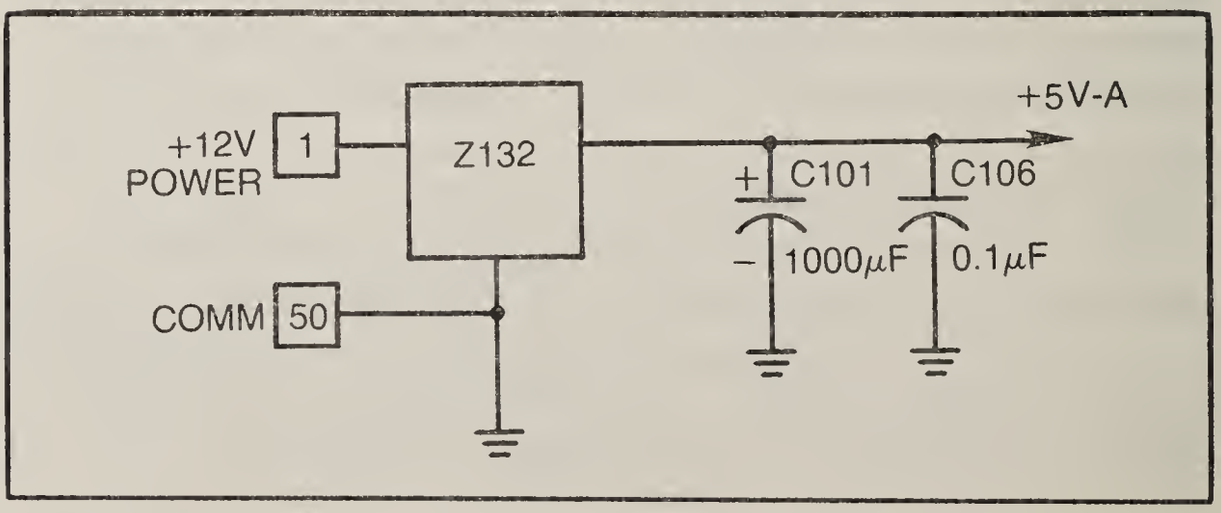


Fig. 6-6. Schematic diagram of the +5V-A power supply.

Table 6-3. CPU board parts list for the select circuits.

ICs
Z103, Z104—74LS367 hex 3-state buffer (Jameco)
Z105 —74LS139 dual 2-line to 4-line decoder (Jameco)
Z108, Z109—74LS138 3-line to 8-line decoder (Jameco)
Z110, Z111—74LS04 hex inverter (Jameco)
Z112 —74LS02 quad 2-input NOR gate (Jameco)
Z132 —5V, 1A regulator, 7805 or LM340T-S (Jameco)

Capacitors
C101—1000μF, 16 WVDC electrolytic (Jameco)
C106—0.1μF mylar (Jameco)

Wire-Wrap IC Sockets
3 ea. 14-pin WW socket (Jameco)
5 ea. 16-pin WW socket (Jameco)

Other Parts
1 ea. TO-220 heat sink (Jameco 291-.36)
1 ea. Vector 8801, 100-pin circuit board (Jameco)
1 ea. 50/100 WW connector (Jameco R681-1)
Vector T46 WW terminal pins (Jameco)
Kynar WW wire, assorted colors

and vertical coordinate $\overline{12}$. The lower left-hand pin of Z109 then goes into hole 40, $\overline{15}$.

Install the wire-wrap sockets for these two ICs in their coordinate-designated places. Then use them as guides for positioning the other six IC sockets. Be sure to use 14-pin sockets for Z110, Z111, and Z112, and 16-pin sockets for Z103, Z105, Z108, and Z109. Tack-solder each IC socket into place at a minimum of two places each.

Insert T46 wire-wrap terminals at the 100-pin connector pads on the CPU board, using the list in Table 6-4 as a guide. Tack-solder each of them into place. You will be using a lot more of these

Table 6-4. Summary of CPU board connectors used for function- and I/O-select tests.

Component Side	Reverse Side
1—+12V 10—A0 11—A1 12—A2 18—A8 19—A9 20—A10 21—A11 50—COMM	K(59—SOD M(61)—RD N(62)—R/W P(63)—$\overline{\text{CSMM}}$ V(68)—R/$\overline{\text{P}}$ W(69)—$\overline{\text{LOAD}}$
100-pin edge connections required for testing the function and I/O-select circuitry	

connections later on, but these are the ones necessary for function- and I/O-select tests.

Complete the assembly of this phase of the job by wire wrapping the circuit according to the pin numbers in Figs. 6-3 and 6-4. Install the IC devices.

Installing and Wiring the 100-pin Connector

Install the 100-pin connector for the CPU board on the card rack assembly. It should be located about 2 inches below the I/O board connector (Fig. 6-7).

Now is the time to wire wrap some of the 100-pin connections on the CPU and I/O boards. Table 6-5 is a list of interconnections necessary for checking out the function- and I/O-select operations. You will be running more wires between these two connectors later, these are the important ones for now.

Wrap a length of wire between pin 1 on the I/O board and pin 1 on the CPU board. This particular connection provides +12V power to both boards. Then wrap together pin 10 on both boards for the A0 line of the address bus, pins 11 for the A1 line, etc. To make things simple for yourself, you might note that the pins you are to interconnect have the same numbers on both connectors.

TESTING AND TROUBLESHOOTING THE SYSTEM

With both main circuit boards fastened securely into their respective sockets, run power supply leads from the auxiliary power supply to the power connections on the 100-pin connectors—+12V to pin 1s and COMM to pin 50s. Turn on the auxiliary power supply and get ready to run some interesting tests.

Table 6-5. Wire-wrap connections between the CPU and I/O boards required for testing purposes.

From Board 100, Pin:	To Board 200, Pin:	Function
1	1	+12V power
10	10	A0
11	11	A1
12	12	A2
18	18	A8
19	19	A9
20	20	A10
21	21	A11
50	50	COMM
K(59)	K(59)	$\overline{\text{SOD}}$
N(62)	N(62)	$\overline{\text{LOAD}}$
V(68)	V(68)	R/$\overline{\text{P}}$

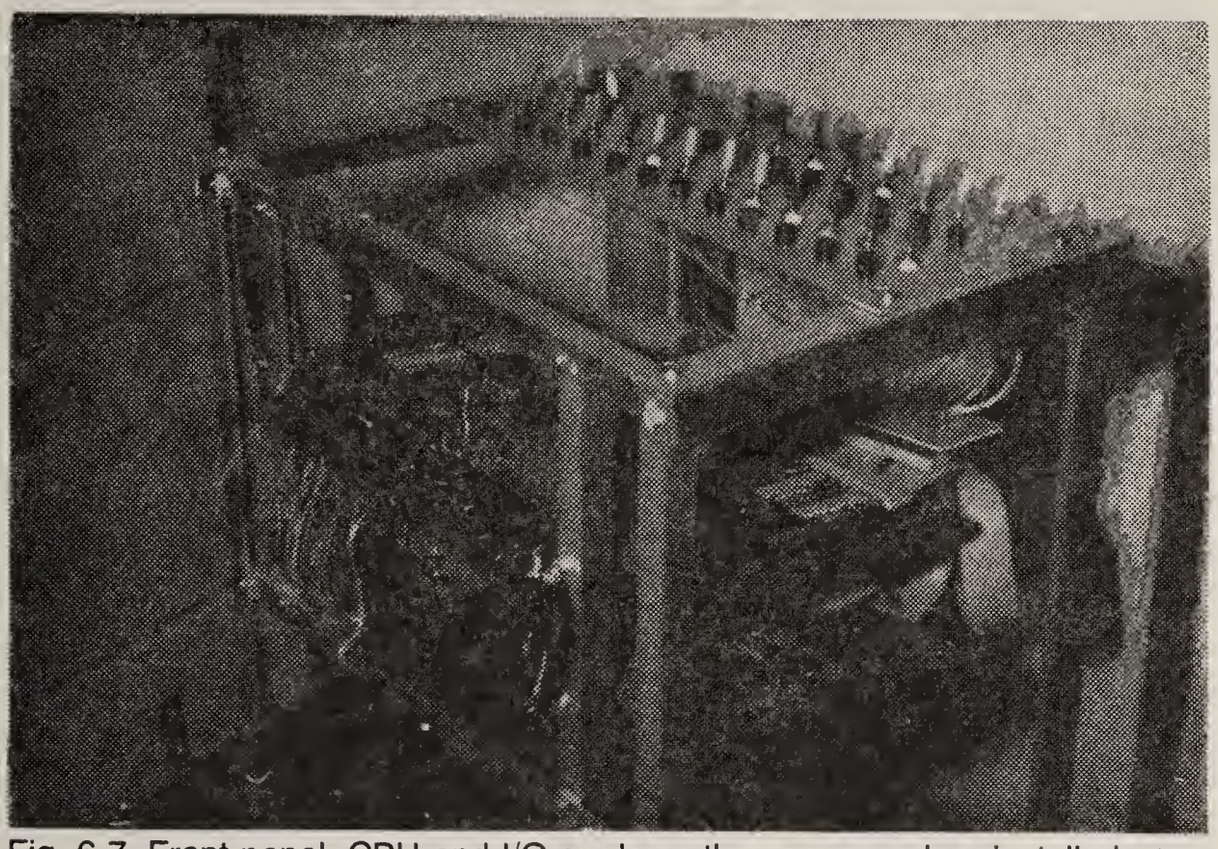


Fig. 6-7. Front panel, CPU and I/O cards as they appear when installed.

Unless clearly specified otherwise, all tests in this section are made at the wire-wrap terminals on the 100-pin connectors. Since like pin numbers are connected together on these two connections, it generally makes no difference whether you use the CPU or I/O connector.

Attach a logic probe or DC voltmeter to pin V and alternate the RUN/PROGRAM switch on the front panel between its two positions. Pin V should show a logic 0 when the switch is in the PROGRAM position, and it should show a logic-1 level whenever the switch is set to the RUN position. If this does not happen, doublecheck the wiring and T46 pin connections at terminal V on both circuit boards. Do not go any further with the testing procedures until you are satisfied things are in good working order to this point.

Then connect the logic probe or voltmeter to pin N and depress the LOAD pushbutton on the front panel. This point should show a logic-0 level when the button is depressed and a logic-1 level when it is *not* depressed. Again, if there are problems, work them out before going any farther.

Check out each of the seven relevant address bus lines by first making certain the system is in the PROGRAM mode (setting the RUN/PROGRAM switch to PROGRAM) and then testing the logic levels at each of the address connections on the card connectors.

To test the AO line, for instance, attach the logic probe or voltmeter to pin 10. With the system in the PROGRAM mode, change the AO switch on the front panel between the "1" and "0" position. Whenever AO is in the "0" position, pin 10 should show a

logic-0 level and when A1 is at the "1" position, pin 10 should be a logic 1. Run the same test for A1 at pin 11, A2 at pin 12, A8 at pin 18, A9 at pin 19, A10 at pin 20, and A11 at pin 21.

The system must be set for the PROGRAM mode to show the desired results. If the system is in the RUN mode, the address lines will be undefined, most likely showing logic-1 levels, no matter what the positions of the address switches might be.

Check out the R/$\overline{W}$ function by attaching the logic probe or voltmeter at pin W and working the LOAD pushbutton. As long as the system is in the PROGRAM mode, pin W should show a logic 1 when the LOAD button is *not* depressed and a logic 0 when LOAD *is* depressed. R/$\overline{W}$ (pin W) is undefined, probably showing logic 1 as long as the system is set up for RUN operations.

The $\overline{RD}$ signal at pin M of the CPU board should show a logic 0 whenever the system is in the PROGRAM mode. It will be undefined, showing a logic 1, by setting the RUN/PROGRAM switch to RUN. At the present time, the SOD signal at pin K of the CPU board ought to show a logic-1 output, no matter what action you take at the front panel.

Be sure to solve any circuit problems as you go along. Ignoring a problem won't make it go away, it only causes even greater problems later on.

The remaining tests for the select circuits are sometimes awkward to perform, simply because it is sometimes hard to get the logic probe or voltmeter to the designated test points. Exercising some care, patience, and creative thinking, however, will go a long way toward carrying out these vital tests.

Check the $\overline{CRAM}$ (program RAM select signal) by attaching the logic probe or voltmeter to pin 12 of Z105. Be sure to use the layout in Fig. 6-5 as a guide. This point should show a logic-1 level under all conditions *except* the following:

- System set for PROGRAM mode
- A10 = 0
- A11 = 0.

Check the $\overline{CROM}$ (optional ROM select signal) by attaching the logic probe to pin 8 of Z110-D. This point should show a logic-1 level under all conditions *except* the following:

- system set for PROGRAM mode
- LOAD pushbutton *not* depressed
- A10 = 1 and A11 = 0

Check the $\overline{CEIO}$ (I/O function select signal) by attaching the logic probe to pin 10 of Z105. This point should show a logic-1 level except when:

- System is set for PROGRAM mode
- A10 = 0
- A11 = 1

These three critical select points should be checked out very carefully. They should always show a clear-cut logic-1 level except when the three special conditions are met at the same time. If you are having trouble with all three of them, look for troubles around Z105, Z112-A, Z110-B, and Z110-C. These components are common to all three function-select signals. The problem might be at address lines A10 and A11; but since you have presumably checked them out already they aren't likely at the heart of the trouble. If you are having some trouble only with the $\overline{\text{CROM}}$ signal, check out Z110-D and Z112-B.

After making certain the main function-select circuitry is operating properly, turn your attention to the I/O-select circuits. Table 6-2 summarizes all the logic conditions. Using this table as a testing and troubleshooting guide is a matter of setting $\overline{\text{CEIO}}$ to logic 0 (PROGRAM mode, A10 = 0, and A11 = 1), and setting R/$\overline{\text{W}}$ to 0 or 1 by means of the LOAD switch. As long as the system remains in the PROGRAM mode, you can establish the eight different combinations of address inputs specified in Table 6-2 by using the corresponding address switches on the front panel.

With this bit of information, Table 6-2, and the schematic in Fig. 6-4 as guides, check the operation of the IOW0 signal by connecting a logic probe or voltmeter to pin 2 of Z111-A and noting that the only time it goes to logic 1 is when $\overline{\text{CEIO}}$, R/$\overline{\text{W}}$, A0, A1, and A2 are simultaneously a logic-0 level. Under any other possible sets of conditions, IOW0 will be at logic 0.

Test all ten I/O select signals in this same fashion, noting whether or not they respond as listed in Table 6-2.

Installing And Testing The Program RAM And Primary Ports

A microprocessor system without any input and output ports is like a car without wheels; it can look very impressive and exhibit a lot of potential, but nothing really significant happens. The Rodney robot system you have constructed to this point doesn't really do much. Oh sure, you can make some lights on the front panel go on and off, and there are a lot of switches to play with.

But the whole project, to this point, is mere foundation work. This chapter paves the way for getting some action from the machine in Chapter 8.

The diagram in Fig. 7-1 illustrates the I/O systems you will be installing as part of the work in this chapter. The diagram shows the system's 12-bit address bus (AO through A11), the system data bus (DO through D7), and some of the I/O and function-select signals derived from circuits you built in Chapter 6.

Specifically, you will be installing the 2-way I/O port, Port 0, and the read-only port, Port 2. The program RAM shown in Fig. 7-1 is not really one of the major ports, but it happens to fit nicely into the scheme of things at this point, so it is included here too.

Port 0 is actually responsible for directing an 8-bit data word to a major portion of Rodney's output devices. Enabled by the IOW0 signal from the I/O select circuitry, Port 0 loads the output buffer (Z115) with an 8-bit data word that ultimately tells the main drive motors what they are supposed to do and, when necessary, enables the motor speed sensing network.

The input portion of Port 0 is enabled by the $\overline{\text{IOR0}}$ signal, accepting an 8-bit data word from the outside world. The data in this case includes four bits telling what the motors are supposed to be doing and two bits that indicate what they are actually doing.

In reality, you will be using the six lower-order data bits, both output and input, at Port 0. One of the remaining bits will remain available for any 1-bit I/O operation you might want to add at some

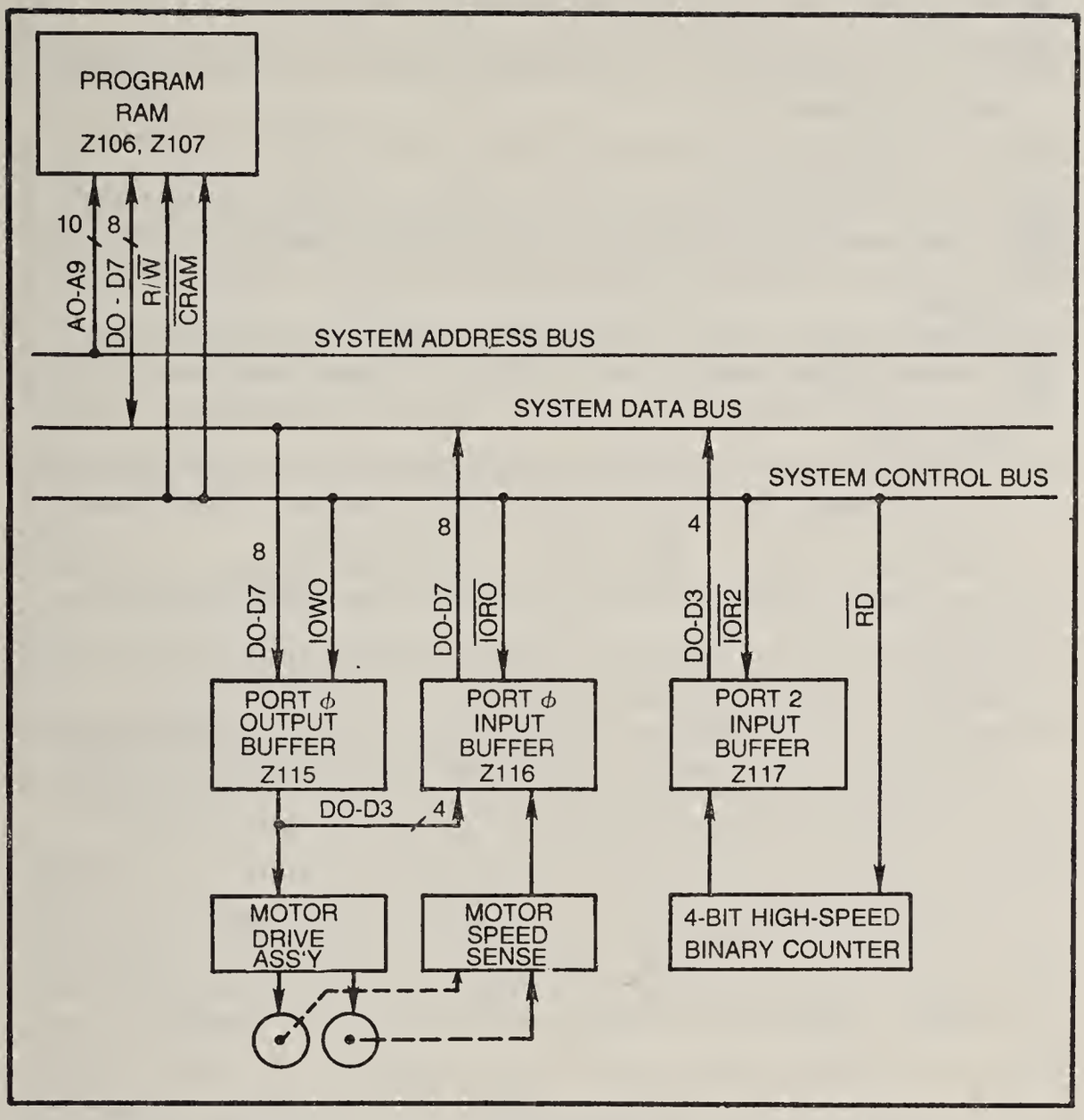


Fig. 7-1. Block diagram of Ports 0 and 2 and the program RAM.

later time. The 8th bit will eventually serve some ancillary functions in the complete Rodney system. In the context of the system's overall view presented in Chapter 2, Port 0 is responsible for the ACTL output and ENVL input operations.

Port 2, also shown in Fig. 7-1, is a read-only port that picks up a quasi-random 4-bit binary number whenever the IOR2 command goes to logic 0. The data in this case is generated by a high-speed binary counter that runs at all times except during a $\overline{RD}$ interval.

The program RAM, composed of Z106 and Z107, is addressed by lines A0 through A9 of the address bus, either writing data on, or reading data from the data bus. Any sort of program RAM operation is enabled by the $\overline{CRAM}$ control, and the status of R/$\overline{W}$ from the control bus determines whether the RAM will read or write data.

Under normal operating conditions, the program RAM is operated by the microprocessor system. However, the RAM, as well as the two primary ports, can be manually operated from the front panel whenever the system is in the PROGRAM mode.

Table 7-1. Glossary of terms for the program RAM and primary I/O ports.

A0 through A11—The system's 12-bit address bus lines
D0 through D7—The system's 8-bit data bus lines
$\overline{\text{CRAM}}$—an active-low signal that enables the program RAM for either reading or writing operations
R/$\overline{\text{W}}$—a control signal that determines whether the program RAM will read (R/W = 1) or write (R/W = 0)
IOWO—an active-high signal that enables Port 0 for writing a new 8-bit ACTL
$\overline{\text{IORO}}$—an active-low signal that enables Port 0 for reading the ENVL status word
$\overline{\text{IOR2}}$—an active-low signal that enables Port 2 for reading a 4-bit random binary number
TSW0 through TSW3—The four binary bits for the Port-2 random binary number
ACT0 through ACT7—the eight low-order bits of the robot's output action code
ENV0 through ENV5—the six lower-order bits of the robot's environment-sensing status word.
LMR, LMF, RMR and RMF—control signals that ultimately determine the direction of motion of the robot's two drive motors: left motor reverse, left motor forward, right motor reverse and right motor forward respectively
FSLR, LSR, FSLL and LSL—command signals that request a testing of the robot's motor speeds: FSLR and LSR for the right motor, and FSLL and LSL for the left motor
SSR and SSL—motor speed sense signals from the motor assemblies: SSR from the right motor, and SSL from the left motor
RMS and LMS—components of the ENVL status word that indicate the stall status of the robot's motors: RMS for the right motor, and LMS for the left motor.
RD—an active-low signal indicating whether or not the system is calling for a read operation (RD = 0 for read, RD = 1 for not read)

Getting into things a bit deeper, take a look at Table 7-1. This is a glossary of terms relevant to the primary I/O schemes featured in this chapter. Some of the terms should seem quite familiar to you because they have been used in the sections you have already studied, constructed, and tested. Other terms explain themselves rather well, but don't be too upset if a few of them don't make any sense at this point.

ADDITIONS TO THE CPU BOARD

The schematics in Figs. 7-2 and 7-3 show the circuits to be added to your existing CPU board, Board 100. Figure 7-4 shows the layout of these components as well as some of the IC devices you have already installed.

The system's program RAM is a static 1024 × 8 Random Access Memory. In other words, it is possible to read or write 1024 eight-bit data words into any addressable location. You will find that this RAM has more storage capability than you will need for even the most complex operating programs.

Since each of the RAM chips handles only four bits of data it is necessary to use two RAMs. Z107 handles the four lower-order data bits (D0 through D3) and Z106 handles the four higher-order bits (D4

through D7). The two ICs are addressed from the same set of 10 address lines (A0 through A9), enabled at the same time by $\overline{\text{CRAM}}$, and both set for either reading or writing operations by a common R/$\overline{\text{W}}$ line.

Storing an 8-bit data word into this RAM circuit is a fairly straightforward process. First select the desired address location where the word is to be stored by setting the corresponding 10-bit number on address-bus line A0 through A9. Then set the word to be stored on the data bus lines, enable the RAM by setting $\overline{\text{CRAM}}$ to logic 0, and finally load the data into the RAM by setting R/$\overline{\text{W}}$ to logic 0. The data word will then be entered into the RAM at the specified address and remain there until (1) a new data word is written into the same address location or (2) there is a loss of DC power to the RAM circuit.

Reading data that has been stored in the RAM at some earlier time is simpler by one step. Specify the desired address location by setting address inputs A0 through A9, then enable the RAM by pulling $\overline{\text{CRAM}}$ down to logic 0. The stored data will appear on the 8-bit data bus as long as R/$\overline{\text{W}}$ remains in its normal, logic-1 state.

All the operations for addressing the program RAM, enabling the RAM, and causing it to store or read out data can be performed rather easily from the system's front-panel assembly. See if you can

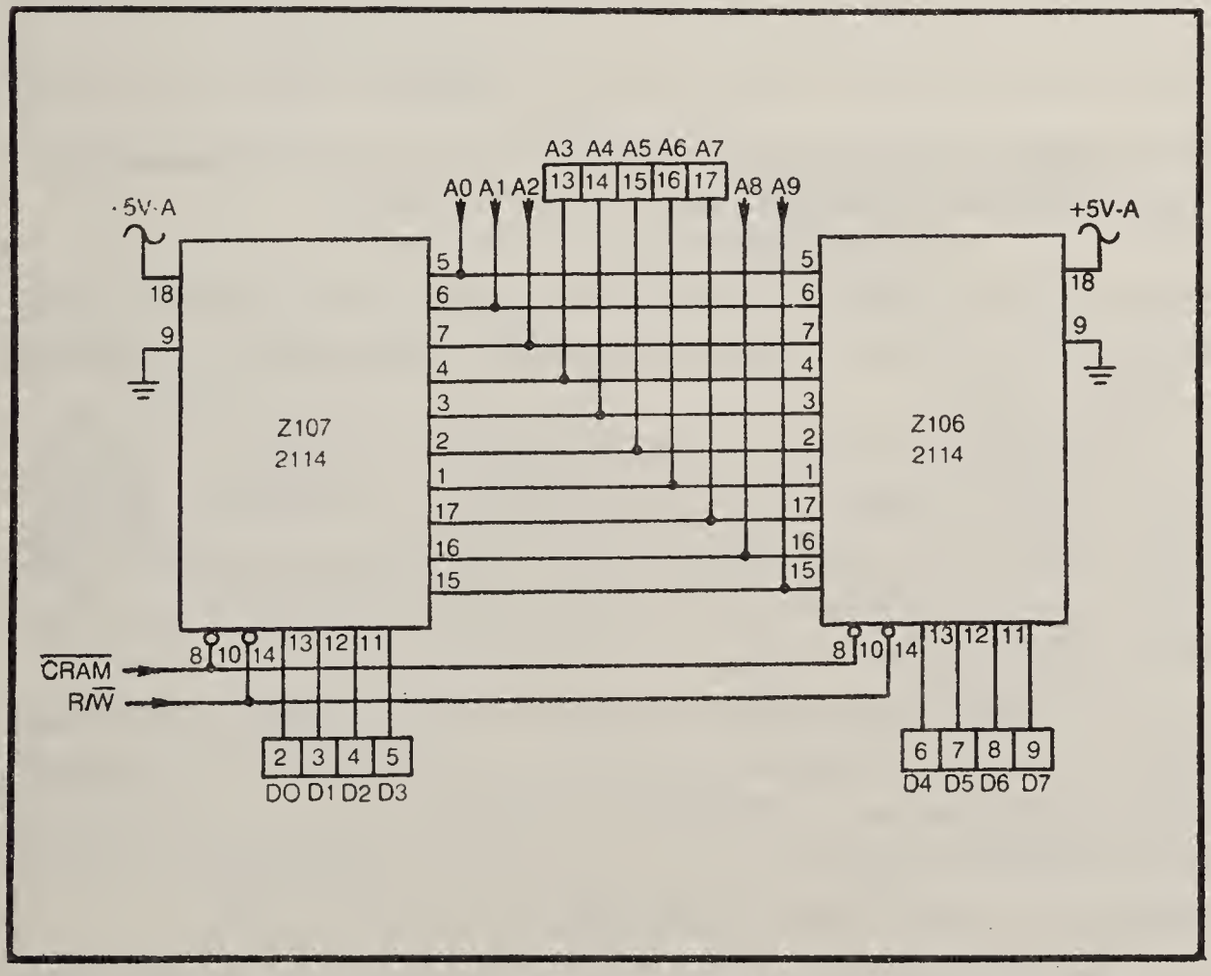


Fig. 7-2. Schematic diagram of the program RAM circuit on the CPU board.

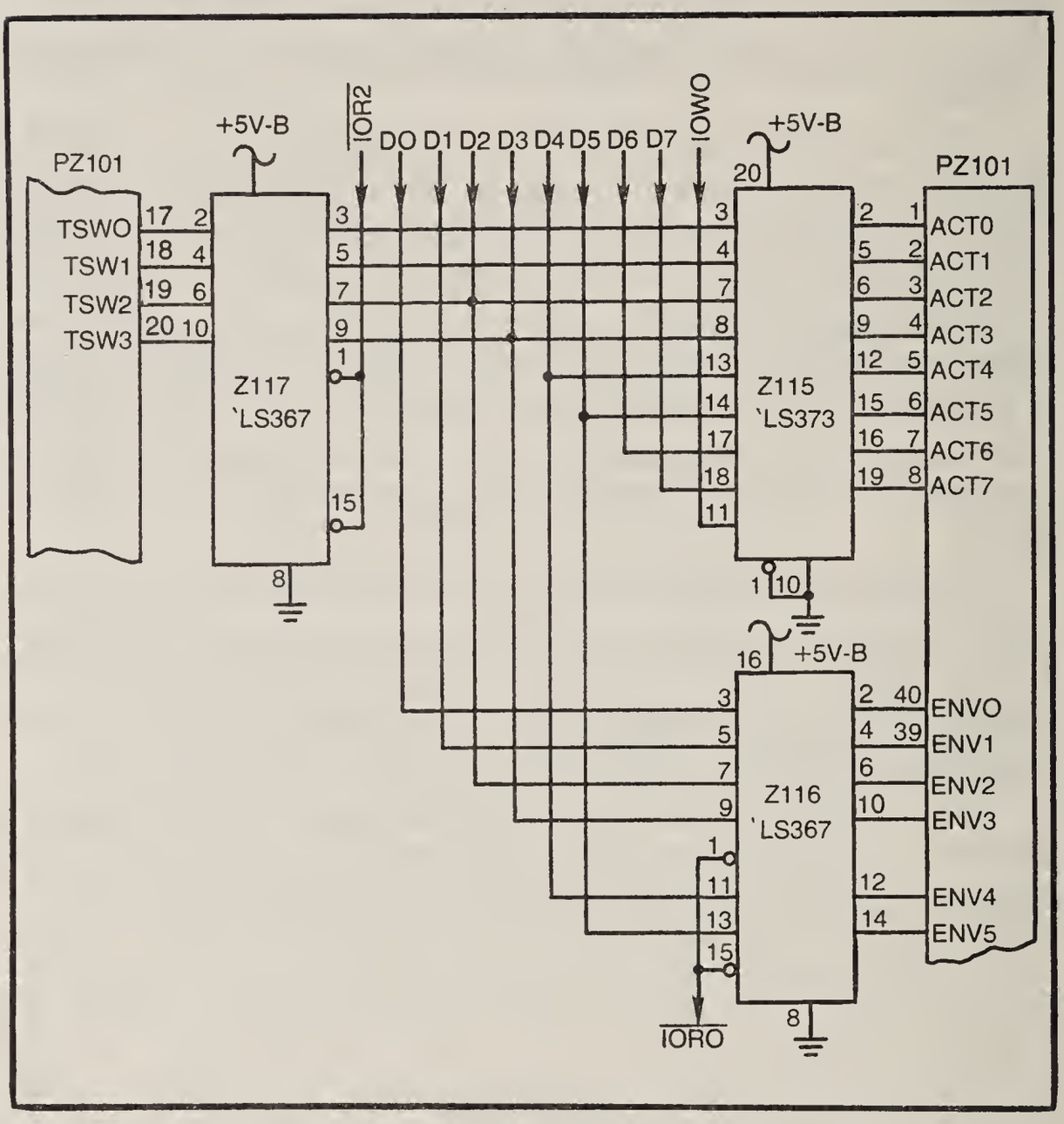


Fig. 7-3. Schematic diagram of Port 0 and Port 2 buffers.

avoid the temptation of running ahead and trying out the RAM system until you have a chance to see what else is going to happen on the CPU board.

The diagram in Fig. 7-3 includes the I/O buffers for Ports 0 and 2. The communication in this case is between the system's data bus and a large, 40-pin connector (PZ101). This 40-pin connector will soon carry I/O information between the CPU and I/O boards via a 40-conductor ribbon cable.

Whenever the system calls for an ACTL output operation, IOW0 enables Z115 so that information on the data bus is entered and latched at the output section of Port 0. The 8-bit data word from Z115 to PZ101 represents motor-direction commands as well as requests for motor-speed information.

The environment, at least the ENVL portion of it, is picked up by Z116 whenever it is enabled by $\overline{\text{IOR0}}$. The information is then applied directly to the data bus as long as $\overline{\text{IOR0}}$ remains at logic 0.

Z116 does not have the latching capability that characterized the output function from Z115. Z116 is simply a 3-state hex buffer.

The Port-2 input buffer works much the same way as the output section of Port 0. Z117, in this case, is enabled whenever $\overline{\text{IOR2}}$ goes to logic 1, placing whatever 4-bit word is present at the TSWR inputs directly onto the four lower-order lines of the system data bus. Eventually, you will be connecting the four TSWR lines (TSW0 through TSW3) to a quasi-random number generator on the I/O board.

Figure 7-4 shows how these additional components should be placed on the CPU board, Board 100. A list of the additional parts appears in Table 7-2.

Install the sockets for Z106, Z107, Z115, Z116, and Z117 into their designated locations. Insert the 40-pin socket (PZ101) into its place just to the right of the 5V regulator you installed in the upper left-hand corner of the board as part of the work in Chapter 6. Tack-solder all the sockets at two or more points.

Note from Figs. 7-2 and 7-3 that the RAMs are powered from the existing +5V-A source, while the port buffers are operated from a +5V-B source. You have to add this second regulator, +5V-B, along with its filter and deglitching capacitors, in the lower right-hand corner of the CPU board. Install the regulator assembly using the

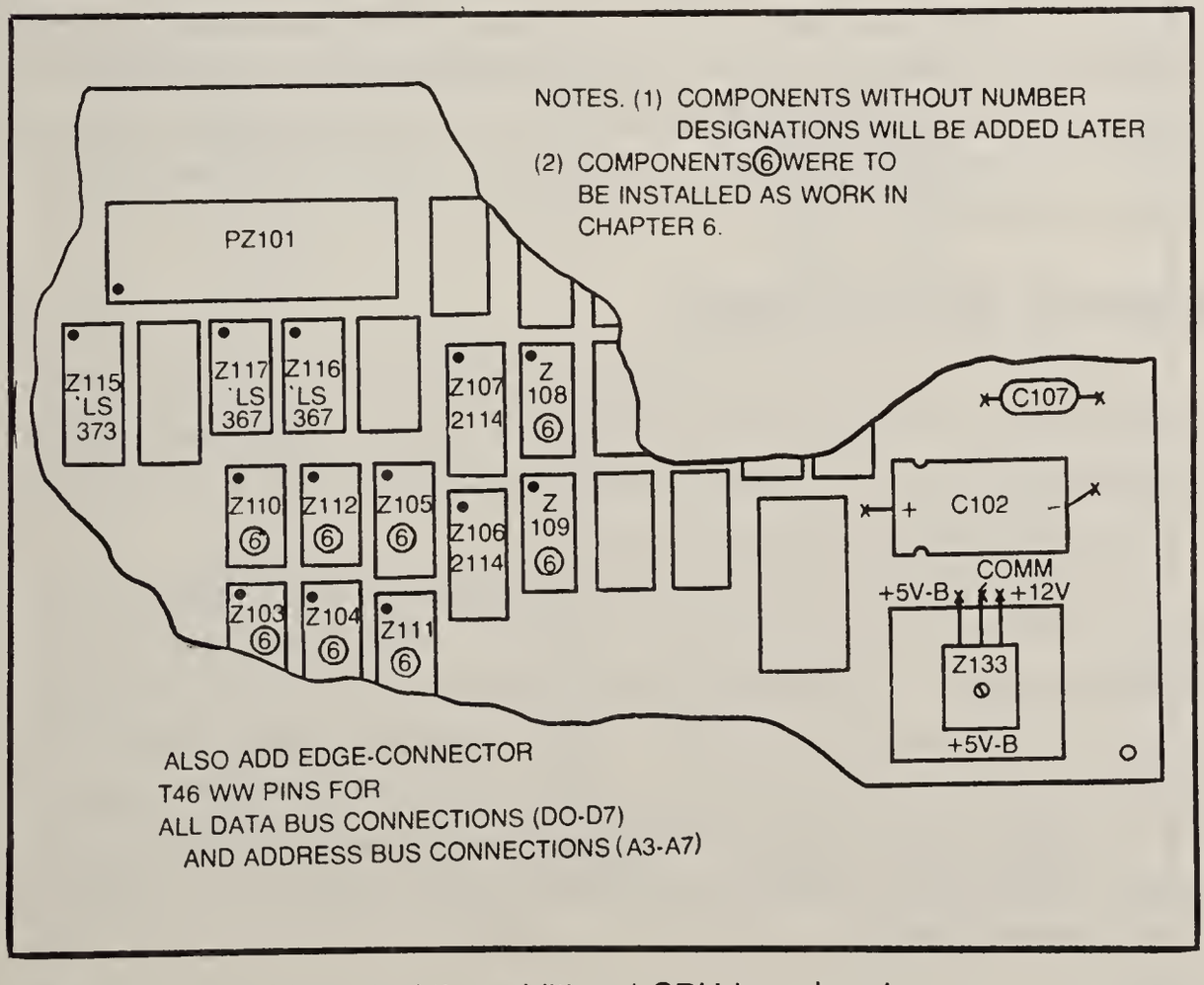


Fig. 7-4. Physical layout of the additional CPU board parts.

Table 7-2. List of additional parts for the CPU board.

ICs
Z106, Z107—2114 1024×4 static RAM (Jameco)
Z115 —74LS373 octal D flip-flop (Jameco)
Z116, Z117—74LS373 hex 3-state buffer (Jameco)
Z133—5V, 1A regulator, 7805 or LM340T-S (Jameco)

Capacitors and Resistors
C102—1000μF, 16 WVDC electrolytic (Jameco)
C107—0.1μF mylar (Jameco)
R250, R251—470k resistor

Wire-Wrap IC Sockets
2 ea. 16-pin WW socket (Jameco)
2 ea. 18-pin WW socket (Jameco)
1 ea. 20-pin WW socket (Jameco)
1 ea. 40-pin WW socket (Jameco)

Other Parts
1 ea. TO-220 heat sink (Jameco 291-.36)
Vector T46 WW terminal pins (Jameco)
PC1—40-pin DIP cable assembly (Jameco DJ40-1-40)
Kynar WW wire, assorted colors

Source: Jameco Electronics
1021 Howard Ave.
San Carlos CA 94070

same procedures and circuit outlined for the +5V-D source on the I/O board. See Fig. 5-12 and the associated technical discussion.

You will also have to add a few more T46 wire-wrap terminals at the edge-connector pads. Table 7-3 shows the connections you should have at the pads for the CPU board when this part of the job is completed. As always, make certain each of the terminals is soldered to its pad, otherwise you will end up with some intermittent signal connections.

ADDITIONS TO THE I/O BOARD

Getting Ports 0 and 2 operational calls for adding a few more components to the I/O board. You will find the additional parts specified in Table 7-4, the necessary schematics in Figs. 7-5, 7-6 and 7-7, and the physical layout in Fig. 7-8.

The I/O board picks up its Port-0 signals from the 40-pin DIP plug and cable assembly connected to PZ201. The circuits in Fig. 7-5 shows the four lower-order data bits from the Port-0 latch going to open-collector inverters in Z215. The motor-controlling outputs of these inverters are tied directly to pads at the board's edge connector. In a later section of this chapter, you will see how these four signals ($\overline{\text{LRM}}$, $\overline{\text{LMF}}$, $\overline{\text{RMR}}$, and $\overline{\text{RMF}}$) are conditioned to operate motor-control relays.

Pins 5 and 6 from PZ201 in Fig. 7-5 feed logic levels through a set of current-limiting resistors (R216 and R215). Ultimately, these signals will operate LEDs in the motor-speed sensing circuit. What is important at this time is to note that the two resistors are located on a device called PR201. PR201 is simply another one of those 24-pin DIP headers we have been using for mounting a lot of resistors on the I/O board. A complete layout for PR201 is illustrated in Fig. 7-7.

The primary purpose of the circuitry in Fig. 7-5 is to condition the six bits of ACTL that will be used for Rodney's primary actions. More will be added later, but these are the essential ones.

The circuit in Fig. 7-6 represents the conditioning circuits for the inputs to Ports 0 and 2. It also includes a wiring diagram for the last regulator to be installed on either of the main circuit boards. Inputs SSR and SSL come from pads c and d on the edge connector of the I/O board. These inputs are actually the output from a pair of phototransistors in the motor-speed sensing circuit. LEDs operated through R215 and R216 in Fig. 7-5 are the light sources for these phototransistors.

Z201-A is an input buffer/amplifier for the SSR, motor-speed sensing signal. It is a linear-type amplifier with a very high gain, designed to translate some rather "indecisive" voltage levels from a

Table 7-3. Summary of 100-pin connections for the CPU-board connector.

Component Side	Reverse Side
1—+12V	K(59)—$\overline{\text{SOD}}$
2—D0	M(61)—$\overline{\text{RD}}$
3—D1	N(62)—R/$\overline{\text{W}}$
4—D2	P(63)—$\overline{\text{CSMM}}$
5—D3	V(68)—R/$\overline{\text{P}}$
6—D4	W(69)—$\overline{\text{LOAD}}$
7—D5	
8—D6	
9—D7	
10—A0	
11—A1	
12—A2	
13—A3	
14—A4	
15—A5	
16—A6	
17—A7	
18—A8	
19—A9	
20—A10	
21—A11	

Table 7-4. List of additional parts for the I/O board.

ICs
Z200—5V, 1A regulator, 7805 or LM340T-S (Jameco)
Z201—LM3900 quad Norton amplifier (Jameco)
Z207—74LS00 quad 2-input NAND (Jameco)
Z208—74LS93 4-bit binary counter (Jameco)
Z215—7406 hex open-collector inverter (Jameco)
Z206—74LS14 hex Schmitt-trigger inverter (Jameco)

Capacitors and Resistors
2 ea. 150-Ohm, ¼W resistor
4 ea. 22k, ¼W resistor
2 ea. 47k, ¼W resistor
8 ea. 10k, ¼W resistor
8 ea. 1M, ¼W resistor

C216—1000μF, 16 WVDC electrolytic (Jameco)
C215—0.1μF mylar capacitor (Jameco)
C217—0.01μF mylar capacitor (Jameco)

Wire-Wrap Sockets
5 ea. 14-pin WW socket (Jameco)
1 ea. 40-pin WW socket (Jameco)
2 ea. 24-pin WW socket (Jameco)

Other Parts
1 ea. TO-220 heat sink (Jameco 291-.36)
Vector T46 terminal pins (Jameco)
2 ea. 24-pin DIP header (Jameco 24-pin HP)

Source: Jameco Electronics
1021 Howard Ave.
San Carlos CA 94070

phototransistor into a clear logic level. Plus, what Z210-A cannot do in terms of cleaning up the linear signal, Z206-A does. Z206-A is a Schmitt-trigger inverting device that ensures a clean, well-defined logic 0 or 1 at the RMS (right motor stall) pin of PZ201.

Now, if you look back to Fig. 7-3, you'll find that pin 36 on the opposite end of this cable assembly is designated ENV4—it is one of six bits included in Rodney's environment sensing word. So the logic status of SSR is a vital sensing input for the system.

The same explanation applies to the SSL input connection in Fig. 7-6. The circuit, in fact, is identical to the SSR circuit just described. In this case, however, the signal represents the operating condition of the left-hand, rather than the right-hand, motor.

Z207 and Z208 make up the high-speed counter circuit that serves the function of a quasi-random number generator. As long as $\overline{RD}$ is at logic 1 (where, indeed, it is most of the time), the three NAND gates oscillate at about 12 MHz. This oscillation is fed to a

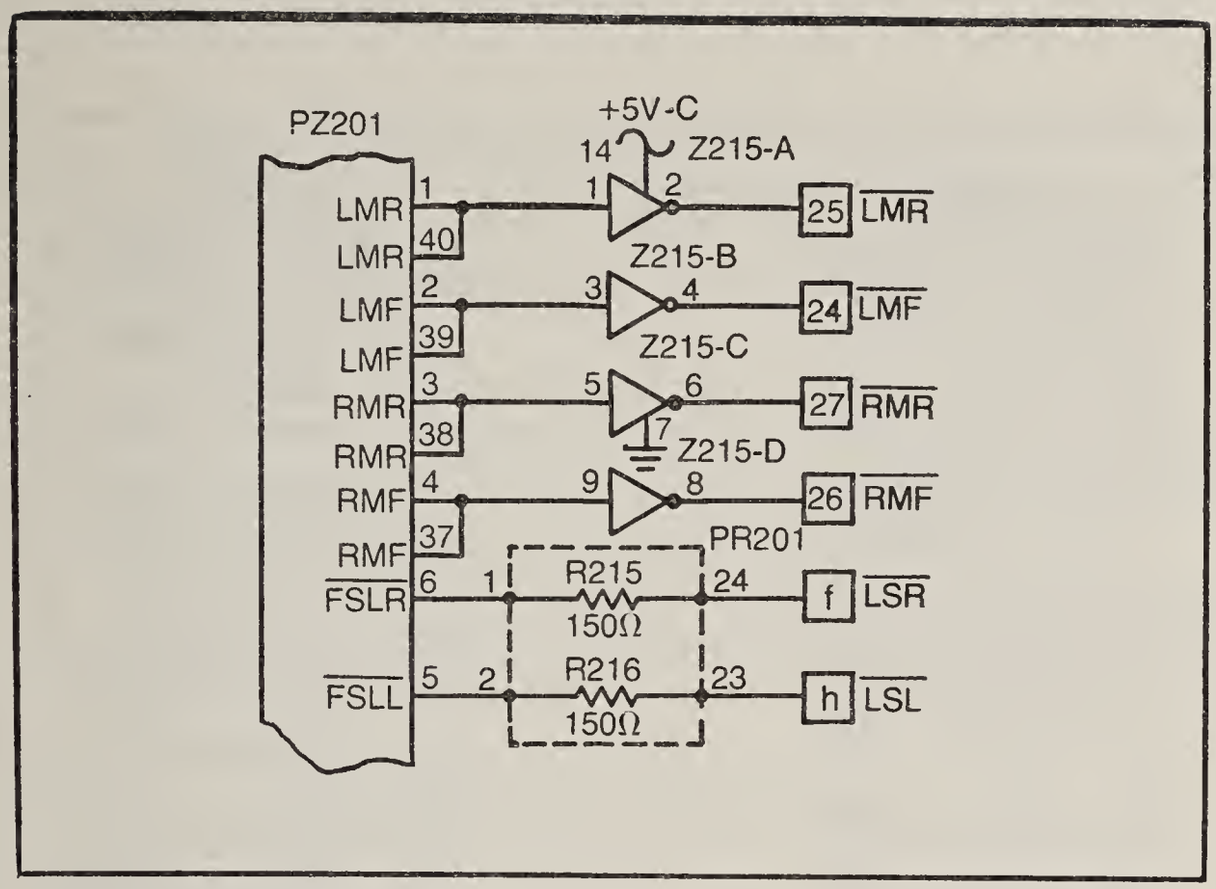


Fig. 7-5. Schematic diagram of the signal-conditioning elements of the Port 0 output.

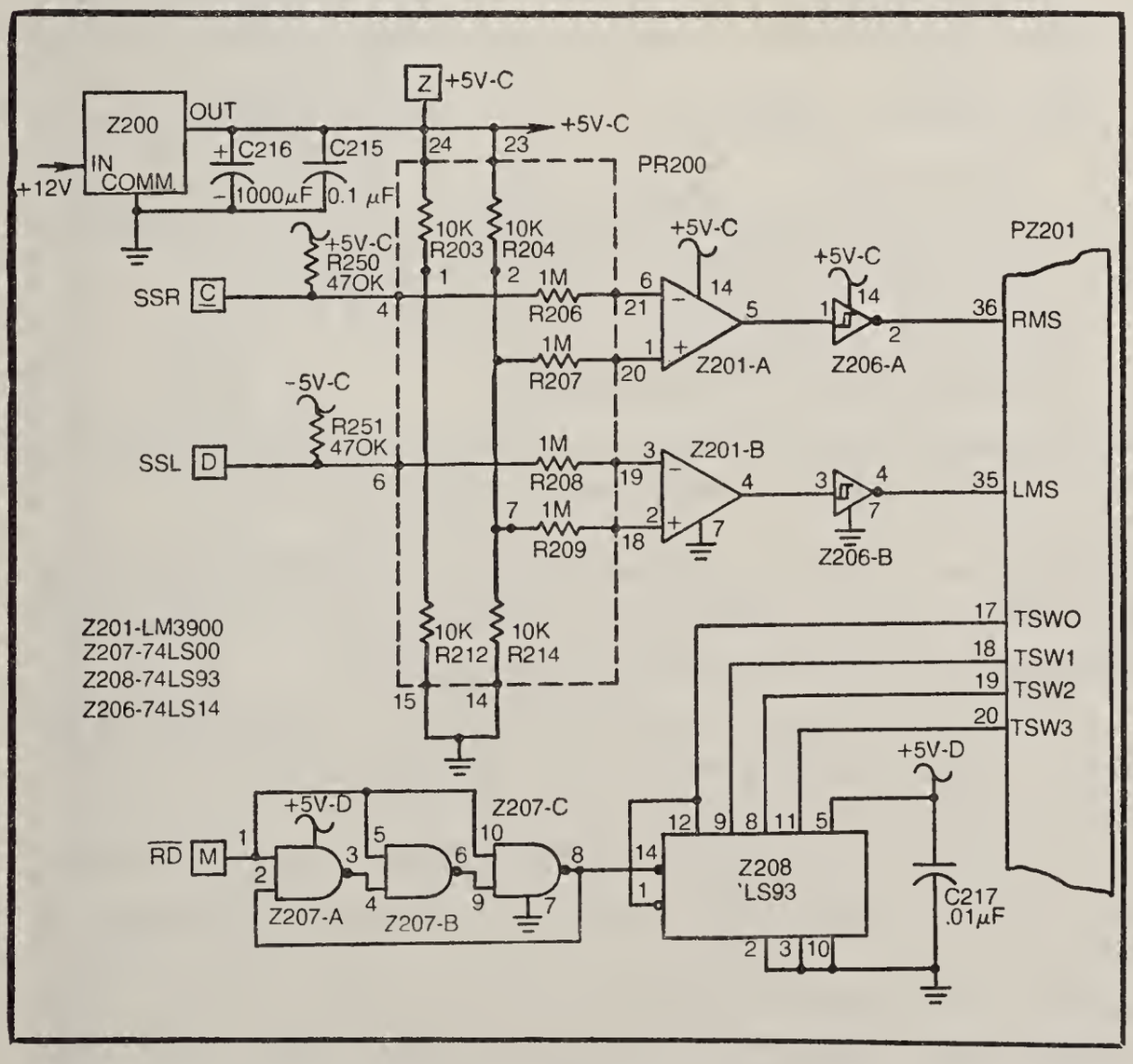


Fig. 7-6. Schematic diagram of the signal-conditioning elements of the Port 0 input and the Port-2 random-number generator.

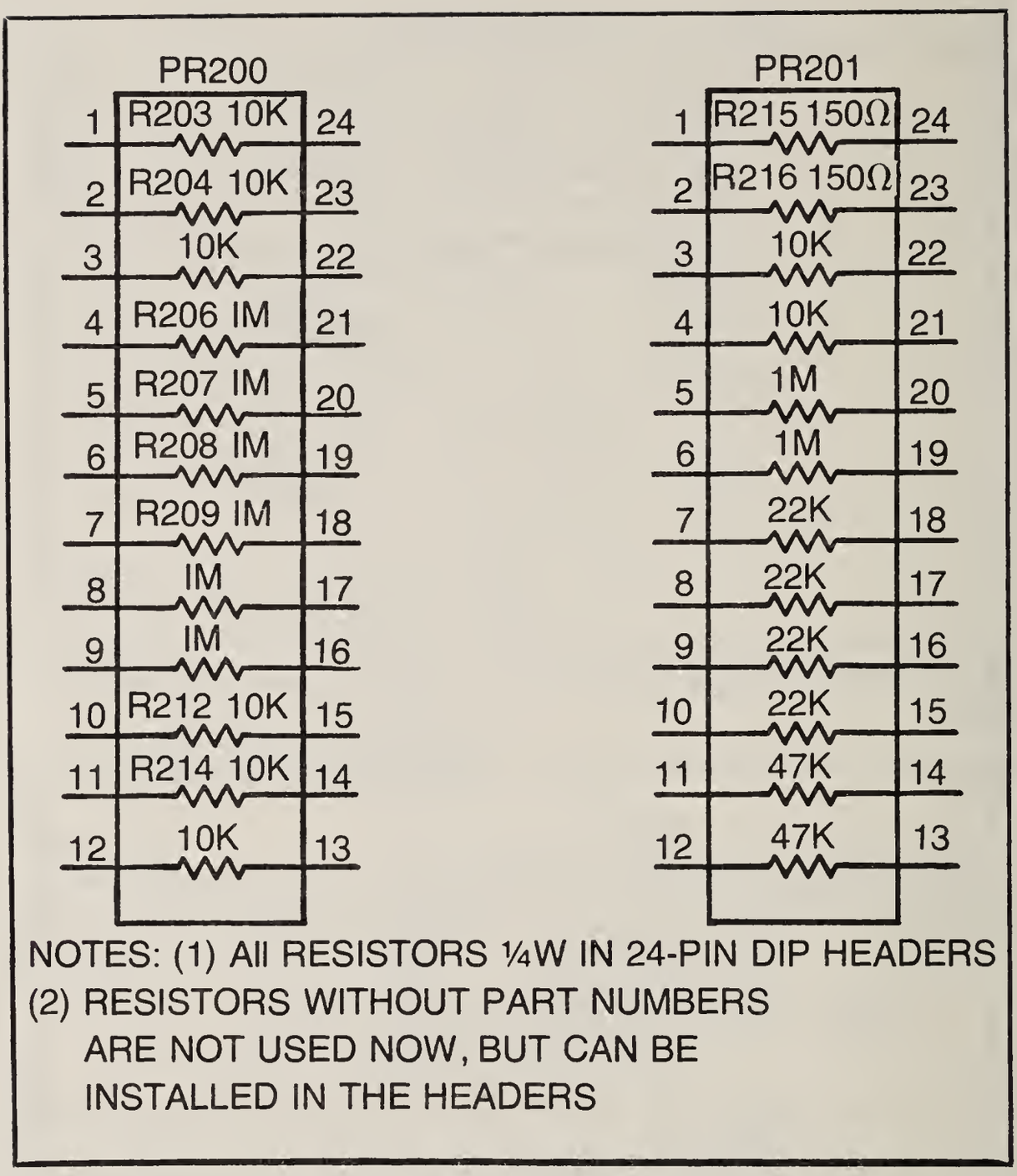


Fig. 7-7. Wiring diagram for resistor headers PR200 and PR201.

clocking input of Z208, an ordinary 4-bit binary counter. This counter thus runs at about 12 MHz, cycling through its 16 possible output combinations, about 750,000 times each second.

Whenever $\overline{RD}$ goes to logic 0, it blocks the oscillations and the counter stops running at one particular 4-bit binary count. This 4-bit number is then used whenever the system calls for a random action.

The data from the counter (Z208) is actually the input information for Port 2 on the CPU board. Since this is a read-only port, it follows that the counter in Fig. 7-6 stops and holds a particular number only when needed during a Port-2 reading operation. The counter stops running for reading operations other than those specified for Port 2, but the counting action is so fast, and the reading operations are so irregular, that Port 2 seems to pick up a random number every time.

Figure 7-7 shows the layout of ¼W resistors on resistor headers PR200 and PR201. Most of the resistors on PR200 are used for biasing the linear amplifiers in Z201, while two of the resistors on PR201 are used for limiting the current to the LEDs in the motor-speed system.

You might want to solder all the resistors to the two headers shown here, even though many of them won't serve any purpose until later in the project. All resistor values are shown, but only those carrying part numbers will be used in this chapter.

Figure 7-8 shows the physical layout of the parts to be added to the I/O board. All of them are located in the upper left-hand corner of the board. The part that is outlined, but does not carry a part

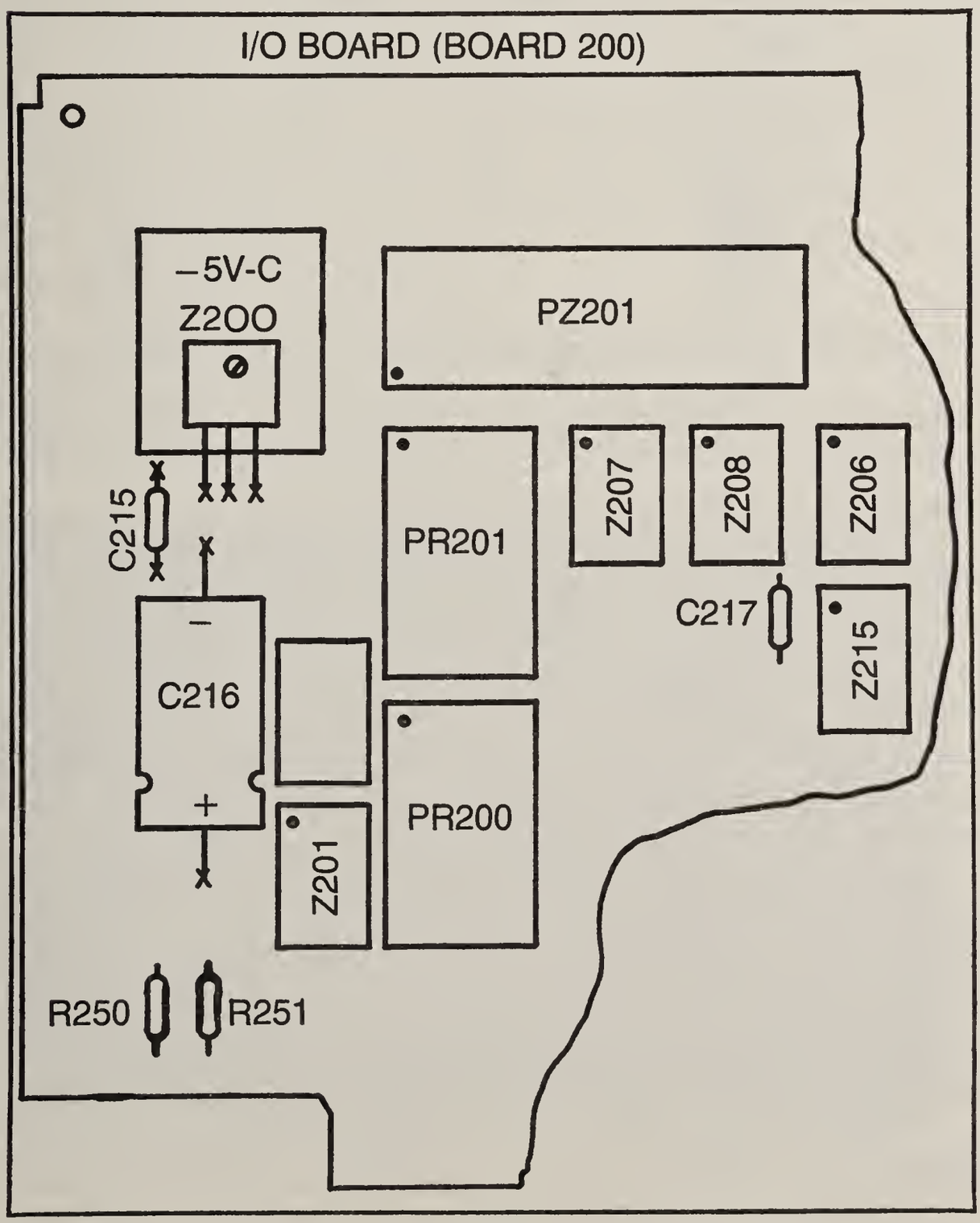


Fig. 7-8. Physical layout of the additional I/O board parts.

Table 7-5. Summary of 100-pin connections for the I/O-board connector.

Component Side	Reverse Side
1—+12V	M(61)—$\overline{RD}$
2—D0	N(62)—R/$\overline{W}$
3—D1	V(68)—R/$\overline{P}$
4—D2	W(69)—$\overline{LOAD}$
5—D3	Z(72)—+5V-C
6—D4	c(75)—SSR
7—D5	d(76)—$\overline{SSL}$
8—D6	f(78)—$\overline{LSR}$
9—D7	h(79)—$\overline{LSL}$
10—A0	
11—A1	
12—A2	
13—A3	
14—A4	
15—A5	
16—A6	
17—A7	
18—A8	
19—A9	
20—A10	
21—$\overline{A11}$	
24—$\overline{LMF}$	
25—$\overline{LMR}$	
26—$\overline{RMF}$	
27—$\overline{RMR}$	
50—COMM	

number, will be added at a later time. Figure 7-9 shows the CPU and I/O boards. Table 7-5 lists all the edge-card pads on the I/O board that should now be wired with T46 wire-wrap pins.

TESTING THE PROGRAM RAM, PORT 0, AND PORT 2

It is possible to work with the program RAM and I/O Ports 0 and 2 from the front panel as long as the system is set up for the PROGRAM mode of operation. In fact, the RAM tests described here use exactly the same procedures you will apply for programming the operational system. The ports, however, will be operated from the microprocessor under normal running conditions.

This will also be your first opportunity to communicate with the system in a hexadecimal code format. If you are already acquainted with the hexadecimal code, you shouldn't have any trouble seeing how it applies here. Just bear in mind that all addressing is done with a 3-character hex number, and all data is input or read out as a 2-character hex number.

Experimenters who are not acquainted with the hex machine-coding scheme will have a chance to learn it here. It is better to learn

it now than to wait until the system is complete and ready to go—you will have enough on your mind at that time.

Hexadecimal Communication With The System

Table 7-6 summarizes the complete hexadecimal code, showing the all-important binary equivalents and incidental decimal equivalents. Note that there are 16 different hexadecimal characters representing binary 0000 through 1111, or decimal 0 through 15.

Whenever a program step calls for entering a pair of hex data characters, 9AH for example, it is to be entered via the data switches as 1001 1010, where the bit on the left is D7 and the bit on the right is D0. (The -H suffix on 9AH merely confirms that the notation is in a hexadecimal format.) By the same token, some address location might be specified as 02FH. The binary version entered on the twelve address switches, in this case, is 0000 0010 1111, where the left-hand bit is A11 and the right-hand bit is A0.

In the PROGRAM mode, information on the address and data busses is read out in binary on the front-panel lamps. For the sake of consistency and convenience, these, too, should be read or recorded in hexadecimal codes. So if something addressing 1000 1110 0010 is showing data 0000 0111, the hex version of the address is then 8E2H and the data is 07H.

The machine works in binary; it wants binary inputs and it outputs codes in binary, but the hexadecimal equivalent of all this binary is easier to understand and record. If you have not already done so, you should take a bit of time to memorize the translation of

Fig. 7-9. The CPU and I/O boards.

Table 7-6. The hexadecimal/binary conversion code.

Hex	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Example 1: 78H data = 0111 1000 data
Example 2: 01AH address = 0000 0001 1010 address

hex to binary and binary to hex. It will certainly save you a lot of grief in the long run.

Checking Out the Program RAM

Apply 12V power to the system from the auxiliary power supply and set the RUN/PROGRAM switch to the PROGRAM position. After that, gain access to the program RAM by setting the address switches anywhere between 000H and 3FFH. Actually, anytime you are in the PROGRAM operating mode, setting A10 and A11 to zero give you access to the program RAM. Those two higher-order bits enable the program RAM system, and the ten lower-order bits are used for establishing the RAM address. Hence, we can say the RAM addressing interval is between 000H and 3FFH.

So set the address switches to 000H, the lowest possible RAM address location, and observe the data at the data lamps. There's no telling what the data will be at this point because nothing has yet been actively stored there. It's "garbage," but the data lamps are showing you what it is anyway.

Now, set the data switches to some interesting pattern of 1s and 0s—AAH, for example. The data lamps should not respond until

you depress the LOAD pushbutton. After depressing the LOAD button, the data lamps should show the same code you loaded from the data switches. This confirms that the data has, indeed, been entered and stored in the program RAM at address 000H.

If any of these operations fail to take place as described here, you will have to check out the wiring of the RAM ICs (Z106 and Z107) on the CPU board. Assuming you conducted all the tests and worked out any problems in the select circuitry (Chapter 6), the trouble has to be in the program RAM chips or its associated address and data wiring.

I have worked around digital electronics for a number of years, and have built, tested, and used a lot of different kinds of circuits. However, I was still excited and rather pleased with myself when I ran these first successful tests on Rodney's RAM.

Play around with the system, first setting up a new address, observing the stored data, and then entering new data. Get yourself accustomed to working with the program RAM in this fashion, because you will be spending a lot of time entering formal programs this way later on.

You will most likely get tired of playing with the program RAM before you have a chance to load some 8-bit data into all 1024 possible address locations. After installing the microprocessor, you can enter a diagnostic program that will doublecheck the function of all those locations within a few seconds.

Checking I/O Port 0

Port 0 is a vital one, even for the simplest Rodney operations. The output function is described as ACTL (low-order action) through the programming sections of this book, and the input functions of Port 0 are designated ENVL (low-order environment).

You can read the ENVL data by simply setting the address switches to 800H while the system is in its PROGRAM mode. ENVL will then appear at the data lamps. When you first access ENVL in this way, it is impossible to say exactly what the data lamps will show. But, if you connect a jumper wire between SSR (pin c) on the I/O board connector and system ground, D5 should show logic 0, connecting SSR to +5V should make D5 show logic 1. The D4 lamp should respond the same way when jumpering the SSL input (pin d) on the I/O board alternately to COMM and +5V.

To obtain better control over things, set the data switches to FFH and depress the LOAD pushbutton. This action enables the ACTL function of the Port 0 circuit. After releasing the LOAD pushbutton, data lamps D0 through D3 should be turned on.

Next, load data 00H. The four lower-order data lamps should then go out. In fact, the four lower-order data lamps should take on whatever code is loaded at their corresponding switches. This is due to the latching action of Z115 on the CPU board and the four turn-around connections at LMR, LMF, RMR, and RMF on PZ201 of the I/O board.

The four higher-order lamps have undefined responses in ENVL sensing. Their inputs are not yet connected. So don't worry about them after checking SSL and SSR as described earlier in this section.

Finally, check the output operation of D4 and D5 at the LSL and LSR connections, pins f and h on the I/O board connector. Connect a logic probe or voltmeter to LSL (pin h), set D4 of the data switches to logic 1, and depress the LOAD button momentarily. As long as the system is still in its PROGRAM mode and addressing Port 0 with an 800H code, the LSL point should show a logic-0 level.

Setting D4 to its "0" position and loading should then cause the LSL test point to show a logic 1. Test LSR in the same fashion, using switch D5 instead of D4. Note that the LSL and LSR responses should have the logic level *opposite* that entered at their respective data switches.

Again, if you have carried out the testing and troubleshooting procedures as carefully as suggested in Chapters 5 and 6, any troubles at this point can be isolated to the few parts just added to the CPU and I/O boards.

Checking The Random-Number Generator

Access the random-number generator at Port 2 by setting up address 802H. As long as the system is in the PROGRAM mode, some binary number should appear on data lamps D0 through D3. The higher-order lamps are not important in Port 2 operations.

Each time you switch the RUN/PROGRAM switch to RUN and then back to PROGRAM again, a different binary number should appear on the four lower-order data lamps, D0 through D3. You should get the impression that you are seeing a randomly generated number each time you make that transition from PROGRAM to RUN and then back to PROGRAM again.

In fact, you can use the Port-2 system as a guessing game. Your chances are exactly 1 in 16 that you can guess the 4-bit binary number at data lamps D0 through D3 each time you make that particular set of PROGRAM-RUN-PROGRAM switch transitions.

If these Port-2 tests aren't working, check the wiring around Z207 and Z208 on the I/O board, as well as the plug and cable assembly between the I/O and CPU boards.

8 Motor-Drive And Power-Distribution Assemblies

You know for sure you're building a robot the first time you hear those drive motors start to whine. That's the sound you should be hearing by the time you complete the work in this chapter.

Here, we are mainly interested in getting the motor drive circuitry installed and checked out. Also, you will check that the subchassis for the motor controls can house the power-distribution system, so you might as well install this section now.

The work in this chapter calls for some more sheet-metal work. If you wish, however, you can substitute the custom chassis with a commercially available one that has about the same dimensions.

MOTOR CONTROL CIRCUIT THEORY

The Rodney robot system uses two independent motors that serve both as the main drive and steering assembly. The motors ought to be identical in all respects; in this way, when they are both running forward at the same speed, Rodney runs across the floor in a straight, forward direction. Running the motors at the same speed, but in opposite directions, causes Rodney to spin on his center axis.

Actually, the 2-motor drive and steering scheme allows nine possible modes of motion, more than enough to get him in and out of just about any conceivable environmental trap. We will discuss these modes of motion in greater detail later in this chapter.

It is sufficient to say at this time that there is a need for turning some relatively high-current motors off and on. The outputs of the control system described thus far are hardly adequate for handling the 2 or 3 amperes of running current, not to mention the 5 or 6 ampere start-up surges. So there is a need for a circuit that steps up the control system's current-driving capability.

Figure 8-1 shows a simplified circuit diagram of such a circuit. Input point M represents one line from the Port 0 ACTL output. Whenever the signal at this level is a logic 1, the motor should run and

LED indicator L1 should turn on. Whenever point M goes to logic 0, the motor and LED should both turn off.

The technical problems between input signal M and the motor involve both voltage and current amplification. The voltage levels at input point M are between 0.5V and 4.5V, while the rated voltage for the motor is 12V. The current drive capability of the circuit at point M is on the order of 20 mA, while the surge current for the motor is on the order of 5A or more.

This voltage and current amplification takes place in two ways. The voltage problem is handled by the open-collector nature of the 7406 inverter. A logic level near OV at M is translated into a logic-high level of about 11.5V at $\overline{M}$. A logic level of about 4.5V at M, on the other hand, is translated into a logic-low level close to 0.5V at $\overline{M}$. So that takes care of the voltage-buffering problem.

Transistor Q is connected as a current amplifier. Its main job is to step up the relatively low drive current from the inverter to a level suitable for driving the coil of the control relay, RL.

Whenever point $\overline{M}$ goes to logic 0, the emitter-base circuit of Q is forward biased, and the emitter-collector circuit acts as a closed switch. Whenever point $\overline{M}$ rises to its logic-1 level, however, Q is biased off, and the emitter-collector circuit acts as an open switch connection.

Putting this all together, it turns out that the relay coil is energized whenever point M goes to logic 1, and de-energized whenever point M goes to logic 0.

The relay in this case is of the double-pole variety, having an LED indicator (L1) connected to one of the poles and the motor connected to the other. Both poles are normally open, so whenever the relay coil is energized by a logic-1 signal level at input M, the LED and motor both turn on. Setting input M to logic 0, on the other hand, de-energizes the relay coil, allows both poles to open and turn off the LED and motor.

The whole point of including the 5V LED indicator circuit is to make it possible to work with the motor control system without having to connect the current-gobbling motors to the power supply. The auxiliary power supply can just barely handle the current for one motor; it cannot reliably drive two motors and the electronics at the same time.

When it is time to run Rodney under his own power, the motors will operate from a fully charged battery, and the auxiliary power supply can be disconnected from the system.

The motor-control tests outlined later in this chapter will be conducted with the motors switched out of the circuit. The only

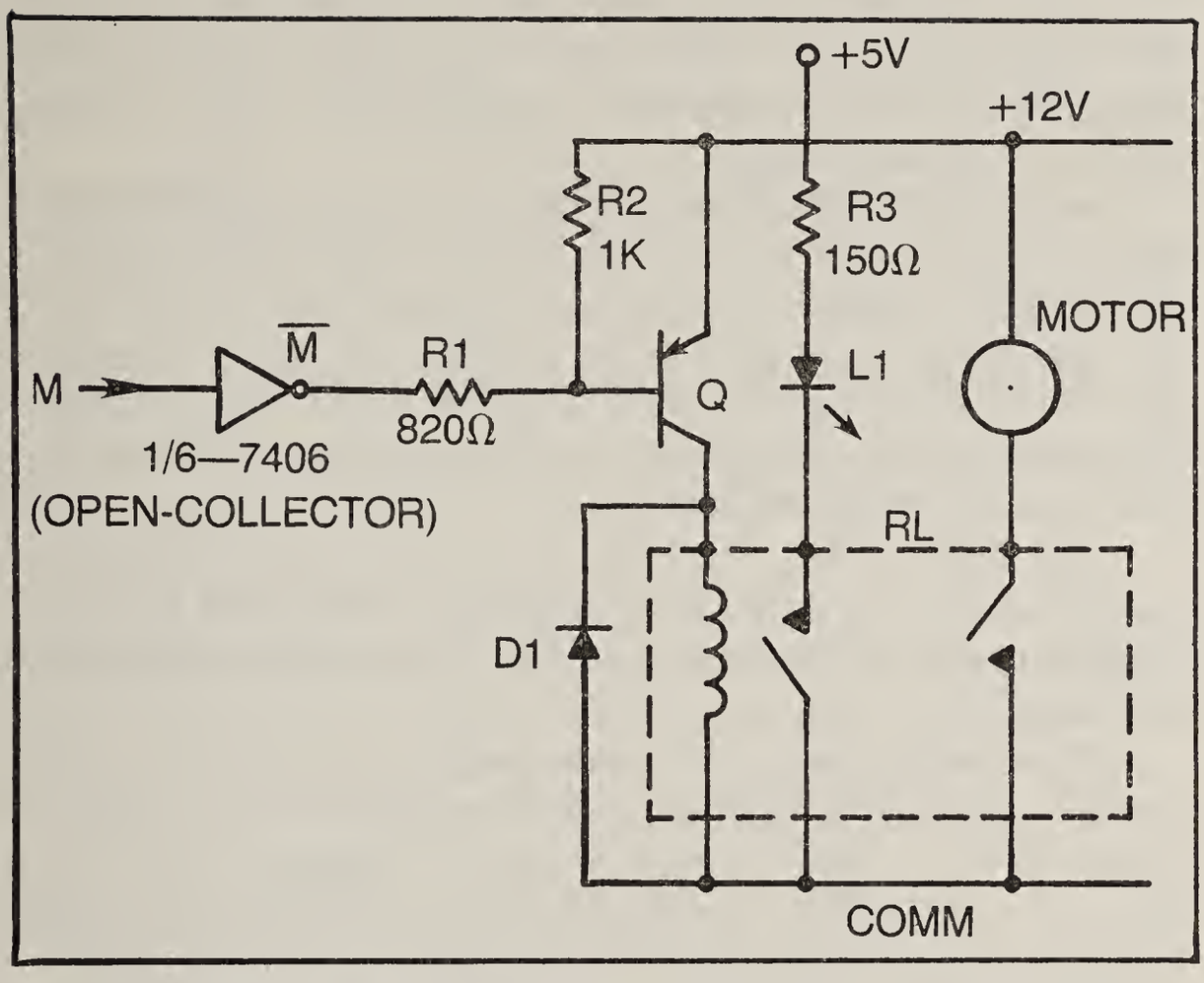


Fig. 8-1. Simplified diagram of a motor circuit.

indication of what the motors ought to be doing is then inferred from the status of the LED operated from the same relay assembly.

Now, I must insert an editorial comment at this point. For those of you who have no initial negative reactions to using relays in a robot system, I suggest you skip over the next few paragraphs.

My own experiences with writing about building robots clearly demonstrate the existence of some widespread misunderstandings about the role of relays in modern electronics technology. I hear from a number of readers who are very anxious to inform me that relays are old fashioned and can be easily replaced with SCRs, triacs, or VMOS devices.

I do not normally take a defensive posture on this matter of using relays in my robot systems simply because my critics generally reflect their own ignorance and lack of practical experience in circuit design.

I like to think I can take a more tutorial position, informing my critics that SCRs and triacs are designed primarily for AC applications, and that it is rather troublesome to use them in DC circuits. VMOS devices certainly have a promising future in this regard, but anyone taking the trouble to look up the prices and delivery times will find the time has not yet arrived for replacing all relays with this sort of semiconductor device.

Funny thing, all these people telling me how old fashioned I am to use slow and inefficient relays, and yet not a single one has shown me an alternate circuit he has actually built; one that works the same way under the same conditions, as economically, reliably, and efficiently as the relay version.

I will admit that there are less expensive relays than the ones I specify in this chapter, but if you use them, I only hope they don't fall apart the first time the machine rams headlong into a solid wall.

The DC motor control scheme in Fig. 8-1 does not include provisions for reversing the direction of the motor. This is a necessary feature of the 2-motor drive and steering system, so it is time to look at another simplified circuit.

The circuit in Fig. 8-2 demonstrates how Rodney's motors are switched to run in two different directions. This circuit uses two separate control relays, one connected to each side of the motor terminals.

When neither relay coil is energized, the contacts are in the positions shown in the diagram. Both terminals of the motor are connected to the positive side of the 12V supply. There is no difference in potential across the motor, so it cannot possibly run. This is one of two different STOP conditions.

The second STOP condition occurs whenever *both* relay coils are energized at the same time. In this case, both motor terminals are pulled down to the negative side of the 12V supply. Again, there is no difference in potential across the motor terminals, and the motor cannot possibly run.

Suppose, however, coil MR is energized and ML is not. The "A" side of the motor is pulled down to the negative side of the 12V source, while the "B" side remains at the positive 12V source. There is thus a 12V difference in potential across the motor terminals, and it runs at full speed in a direction determined by electron flow from terminal "A" to "B." The motor runs at full speed, but in the opposite direction, whenever ML is energized and MR is not.

Indeed, one can suggest simpler and less expensive relay circuits for controlling the motor, but these other versions suffer one of two disadvantages—either the motor cannot be stopped, or there is a distinct possibility of creating a dead short across the power source.

The actual motor control circuit for the Rodney system is a combination of the circuits in Figs. 8-1 and 8-2. There are two identical motors, each one operated from a pair of relays, as shown in Fig. 8-2, and each relay is driven by a transistor circuit such as the one illustrated in Fig. 8-1.

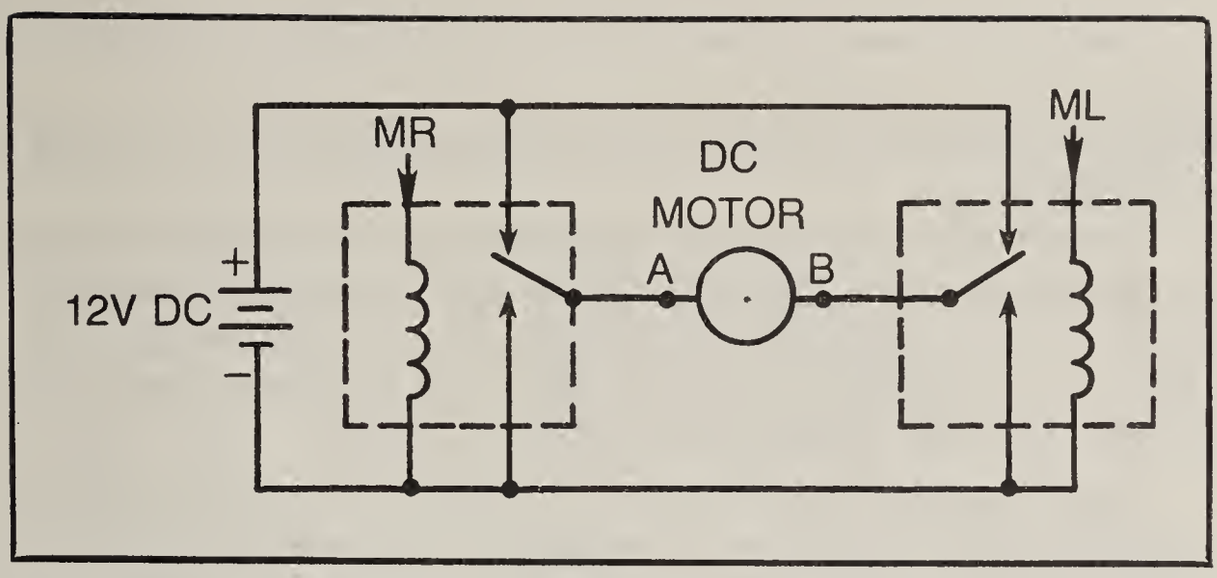


Fig. 8-2. Simplified diagram of a motor direction control circuit.

Figure 8-3 is a complete schematic diagram of the motor drive circuitry. It is shown here only for purposes of understanding how the circuit works and, perhaps, troubleshooting it at a later time. In actual practice, the components will be divided between an aluminum sub-chassis and a small printed-circuit board.

RODNEY'S RESPONSES TO MOTOR-CONTROL SIGNALS

The circuit in Fig. 8-3 shows the motor-control circuit receiving four different logic levels from the Port 0 ACTL output. Working in this way from four different binary bits, there are 16 different combinations of input logic levels. The robot responses are summarized in Table 8-1.

It turns out, however, that some of the robot responses occur under more than one set of binary inputs. There are just nine

Table 8-1. Summary of motor motion codes and motor responses.

Decimal	Hex	RMF RMR LMF LMR	RIGHT MOTOR ACTION	LEFT MOTOR ACTION	ROBOT RESPONSE
0	0	0 0 0 0	OFF	OFF	STOP
1	1	0 0 0 1	OFF	FORWARD	FORWARD WITH RIGHT TURN
2	2	0 0 1 0	OFF	REVERSE	REVERSE WITH RIGHT TURN
3	3	0 0 1 1	OFF	OFF	STOP
4	4	0 1 0 0	FORWARD	OFF	FORWARD WITH LEFT TURN
5	5	0 1 0 1	FORWARD	FORWARD	FORWARD STRAIGHT
6	6	0 1 1 0	FORWARD	REVERSE	CCW SPIN
7	7	0 1 1 1	FORWARD	OFF	FORWARD WITH LEFT TURN
8	8	1 0 0 0	REVERSE	OFF	REVERSE WITH LEFT TURN
9	9	1 0 0 1	REVERSE	FORWARD	CW SPIN
10	A	1 0 1 0	REVERSE	REVERSE	REVERSE STRAIGHT
11	B	1 0 1 1	REVERSE	OFF	REVERSE WITH LEFT TURN
12	C	1 1 0 0	OFF	OFF	STOP
13	D	1 1 0 1	OFF	FORWARD	FORWARD WITH RIGHT TURN
14	E	1 1 1 0	OFF	REVERSE	REVERSE WITH RIGHT TURN
15	F	1 1 1 1	OFF	OFF	STOP

different responses, but they are more than adequate for a working robot system.

So maybe Rodney wants to make a left-hand turn while running in a forward direction. This maneuver is simply a matter of stopping the left drive motor and running the right drive motor forward at full speed. In other words, load a 4H or 7H at the low-order portion of the data bus while enabling ACTL (Port 0 output).

Before completing the work in this chapter, you will be able to test out the motor responses by loading motor-direction codes from the front panel assembly. In fact, you will be able to store a sequence of motor codes in the program RAM and get the motors to respond as you work the address switches. Then one day soon, the microprocessor will be pulling these 4-bit motion codes from the random-number generator or main memory in an automatic fashion.

MOTOR SPEED SENSE CIRCUIT

In a philosophical sense, the motor speed sensing circuit is one of the most critical parts of the Rodney scheme. It is part of a control system that senses a stall condition—it informs the microprocessor that Rodney has made contact with an immovable object of some sort.

Actually, the motor speed sensing circuit tells the system whether or not the motors are running when, indeed, they should be running. If one of the drive motors is supposed to be running at full speed (whether forward or in reverse normally isn't relevant) and that particular motor is not running, the motor speed sensing circuit is the first to know about it.

The motor speed sensing circuit, as described in this chapter, cannot make any logical decisions regarding the status of the motors. The decision-making operations will be done with microprocessor software, and the hardware is capable of doing little more than indicating whether the motor is running or not.

Figure 8-3 shows the general schematic for the motor speed sensing circuit. A simplified physical layout in Fig. 8-3A shows an LED light source and a phototransistor mounted on opposite sides of the gearmotor housing. If you use the gearmotor assemblies specified in this book, it so happens that one of the larger gears has a set of rather large holes drilled through it.

The LED and phototransistor are arranged so that as the motor turns the gear, light from the LED falls onto the phototransistor or is blocked by the space between the holes in the gear. If the motor and gear are turning, the phototransistor sees a blinking light from the LED. If the motor and gear assembly are stopped for any reason, the

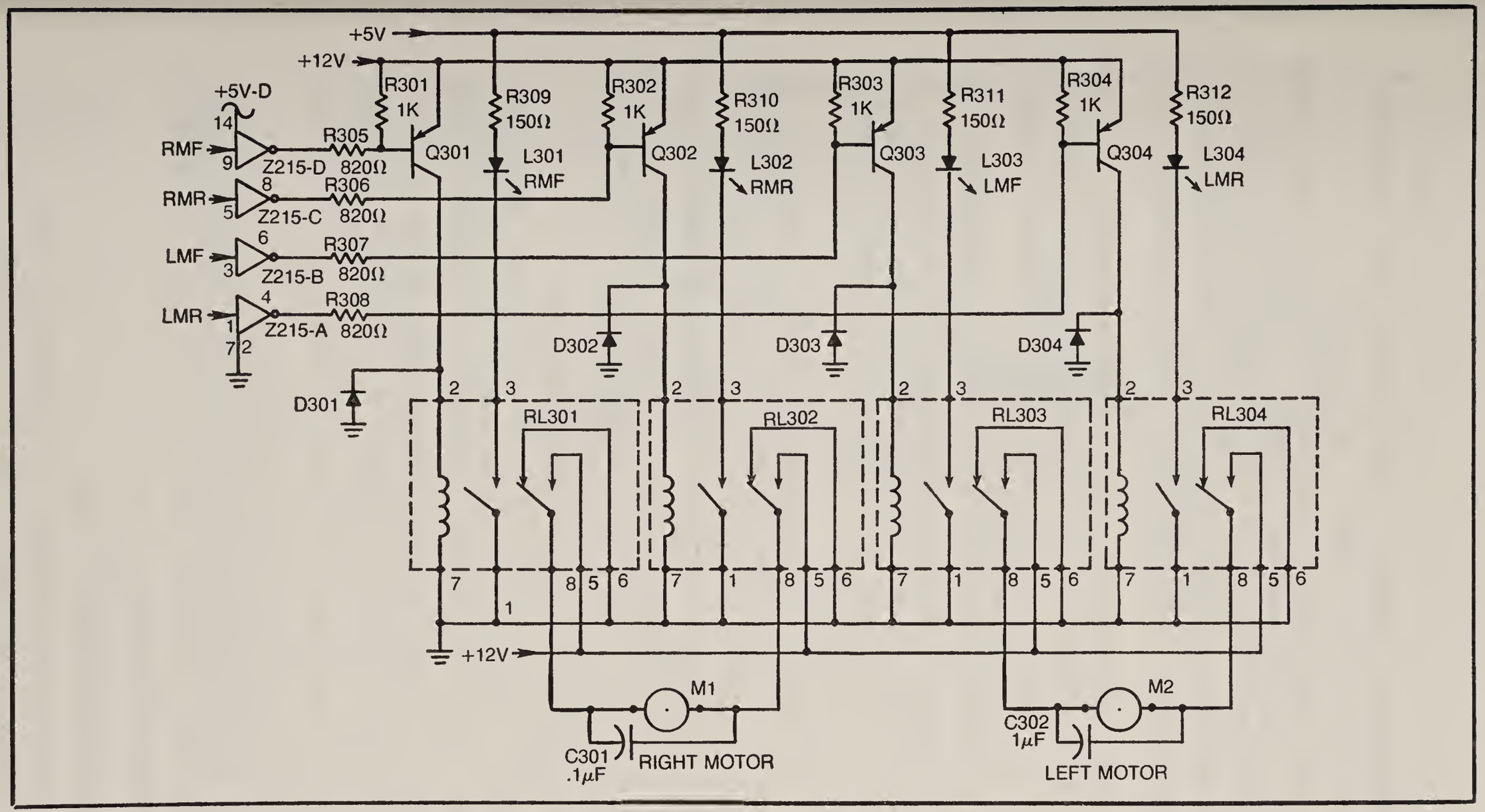


Fig. 8-3. Composite schematic of Rodney's motor control circuit.

phototransistor no longer sees a blinking light. Rather, it sees either a steady light or none at all.

The circuit in Fig. 8-4B shows how this motor speed sensing circuit is interfaced with the electronic control system. LED L1 is normally turned off by a logic-1 level at FSLR, one of the ACTL bits from output Port 0. Whenever it is time to take a look at the motor status, however, FSLR is set to logic 0 and L1 responds by turning on.

Since one of the gears in the gearbox actually rests between the LED and the phototransistor, the phototransistor either "sees" that light, or it doesn't, depending on whether or not one of the gear's holes is lined up between these two devices.

If light does indeed pass from L1 to the phototransistor, the phototransistor conducts and pulls point SSR down toward the COMM potential. If, on the other hand, the gear is blocking the light from L1, PT-1 remains in its switched-off condition, and SSR remains very close to +5V.

The changes in the conduction of the phototransistor, as reflected by a change in the voltage level at SSR, are amplified by operational amplifier Z210-A. The logic levels are inverted by this amplification process, but Schmitt-trigger inverter Z206-A "cleans up" the signal and inverts it again to its "upright" configuration. The RMS (Right Motor Stall) output thus goes to logic 1 whenever there is no light falling on the phototransistor, and it snaps to logic 0 whenever light *is* falling on the phototransistor.

This is only an interfacing circuit, and it carries out no logical operations. It is the microprocessor software that turns on the lamp at the right time, and then interprets the responses at RMS. Actually, the software will be looking for light-to-dark changes. If RMS is showing logic level changes at a specified rate, the system will know the motor is running. If the changes in logic levels from RMS are either too slow or do not exist at all, either the LED isn't switched on or else the motor is stalled.

There are two such motors in the Rodney system, so there are to be two set-ups identical to that in Fig. 8-4. The relevant part numbers and specifications will be included with the main wiring diagrams in this chapter.

POWER-DISTRIBUTION SECTION

Figure 8-5 shows the basic power-distribution scheme for Rodney. The circuit will appear in a somewhat different fashion later on, This one is intended only for purposes of showing how it works.

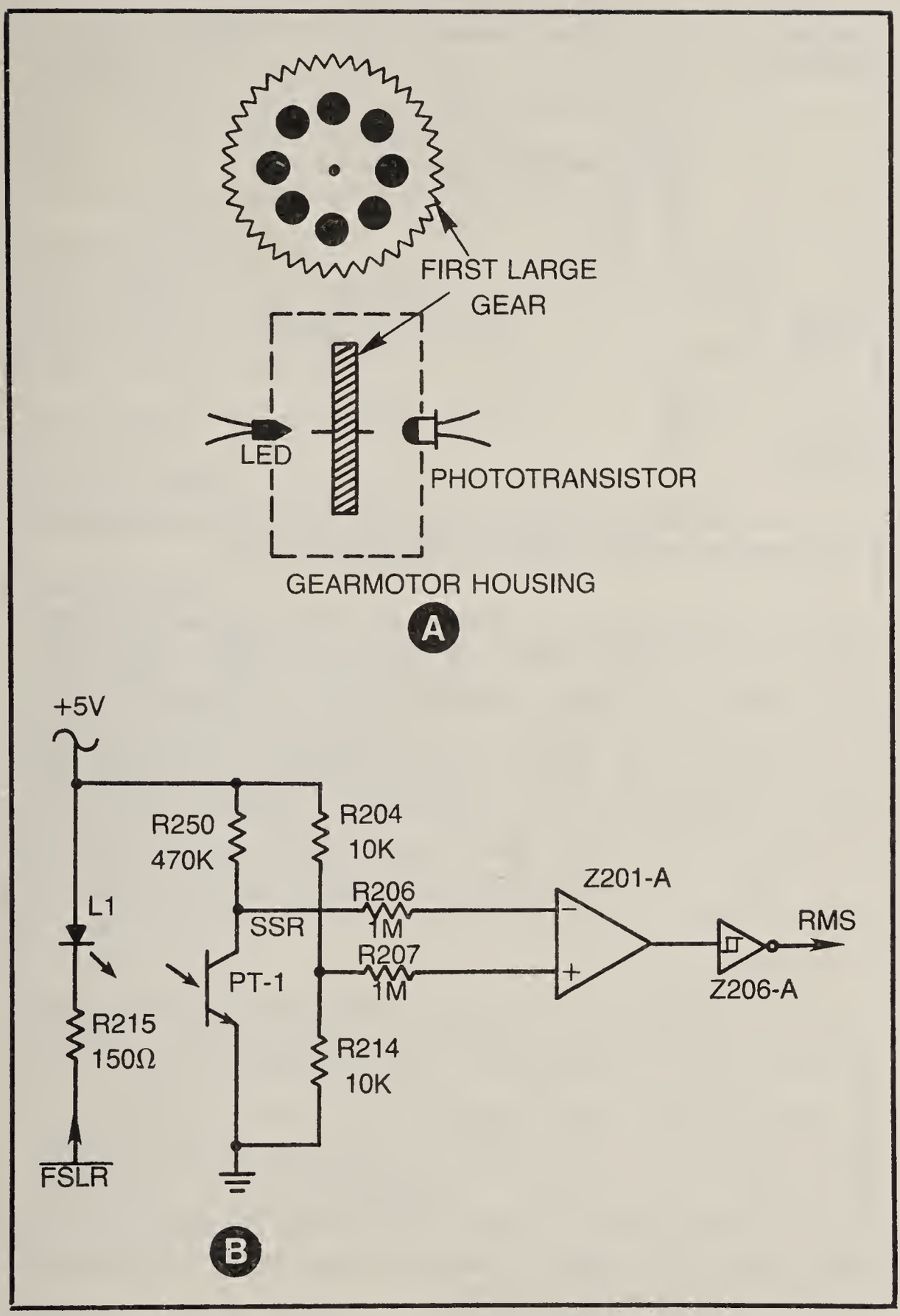


Fig. 8-4. The speed or motion-sensing circuit. (A) Mechanical interface with the motor gear assembly. (B) Schematic of the speed/motion sensing circuit.

The circuit as shown here has three main power connections: the battery charger inputs, the +12V auxiliary power supply inputs, and the main battery pack itself.

Whenever Rodney is in free and full operation, there are no connections to the battery charger and auxiliary power supply terminals. The whole system operates from the main battery. Assum-

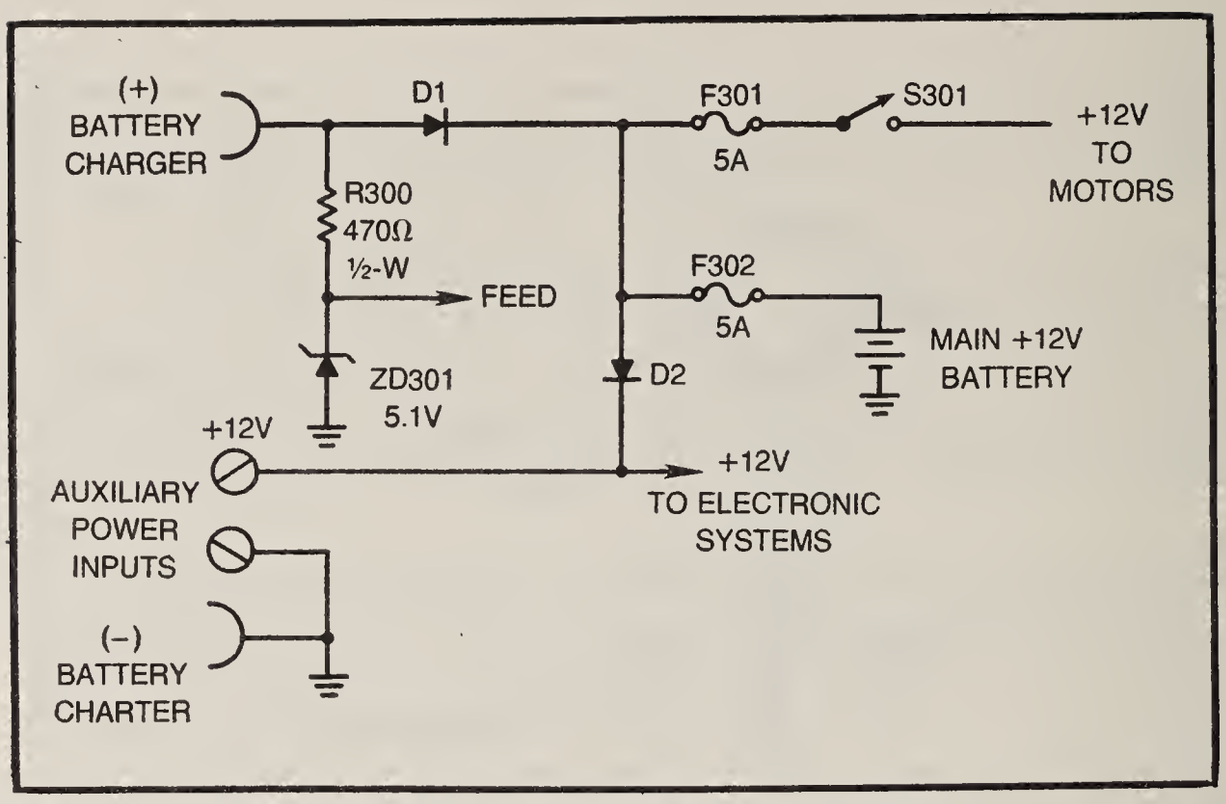


Fig. 8-5. Simplified diagram of the power-distribution system.

ing switch S301 is closed, main battery power reaches the motor-control circuit through fuses F302 and F301. Since diode D2 is forward biased under this particular set of circumstances, main battery power passes through it to the electronic system as well. Note, however, that diode D1 is reverse biased as long as the system is running only from the main battery. This is evident by the fact that the FEED signal will be undefined, or at least NOT at +5.1V. Later on, you will find that D1 also prevents accidental short circuits between the battery-charger terminals while Rodney is running around the floor.

Now, when the battery charger is connected to the charging terminals specified in Fig. 8-5, D1 is forward biased to allow charging current to reach the main battery. At the same time, voltage from the charger allows a zener-regulated 5.1V level to appear at the FEED terminal. This FEED signal is one of the bits in the ENVL, Port-0 input to the data bus. The presence of FEED information informs the microprocessor system that Rodney is connected to the battery charger. The battery charger can also supply 12V power to the electronic system through D2.

Connecting the auxiliary power supply to the system changes things around a bit. If the battery charger and auxiliary power supply are connected to the system at the same time, diode D2 will be reversed biased. In this case, the battery charger will feed only the motors and battery, while the auxiliary power supply will feed only the electronic system. It is important to isolate the auxiliary power supply from the battery and motors because it cannot handle the

Table 8-2. Parts list for the power distribution and motor control circuit. See Fig. 8-11 for additional parts required for the sensing system.

Semiconductors
Q301 - Q304 4 ea. 2N4402 PNP transistor (Jameco)
D301 - D304 4 ea. 1N458 rectifier diode (Jameco)
L301 - L304 4 ea. LED (Jameco XC556R)
ZD301 5.1V, 400 mW zener diode, 1N751 (Jameco)
BR300 6A, 50V bridge rectifier (RS #126-1180)

Resistors and Capacitors
R301 - R304 4 ea. 1k, ¼W resistor
R305 - R308 4 ea. 820 Ohm, ¼W resistor
R309 - R312 4 ea. 150 Ohm, ¼W resistor
R300 470 Ohm, ½W resistor

C301, C302 0.1μF mylar capacitor (Jameco)

Relays and Motors
RL301 - RL304 12V DPDT relay, 10A contacts (RS# 275-208)
M1, M2 2 ea. 12 VDC motors (Surplus Center #5-936)

Other Components
S301—SPST switch (RS #275-602)
TS300, TS301—2ea. 8-terminal barrier strip (RS #274-670)
18 ea. Vector T46 wire-wrap terminals
F301, F302—5A, slow-blow fuses (RS #270-1287)
2 ea. fuseholder (Jameco HKP)
4 ea. Cliplite LED mountings (Jameco)
Wire: AWG #18 stranded, AWG #22 solid hook-up wire, Kynar W-W wire
1 pc. 3-in. × 4-in. single-side PC stock
1 pc. 9½-in. × 7-in. aluminum stock
4 ea. transistor mounting sockets (RS #276-548)

RS = Radio Shack

Jameco = Jameco Electronics
1021 Howard Ave.
San Carlos CA 94070

Surplus Center = Surplus Center
P.O. Box 82209
Lincoln NE 68501

current demand of either circuit. If the auxiliary power supply is connected to the system and the battery charger is not, only the electronic system will be receiving power. This is a desirable situation in the early going where charging the battery and running the motors are unimportant.

There is yet another vital reason for using this particular diode logic scheme between the battery charger and auxiliary power

supply. Recall that the program RAM and main memory systems can be scrambled hopelessly if there is an interruption of supply power. The auxiliary power supply can be used as an emergency back-up system if anything starts going wrong with the battery or battery charger.

ASSEMBLING THE MOTOR CONTROL AND POWER PANEL

The power panel houses most of the circuitry described so far in this chapter. This includes the motor relays, fuses, motor status indicator lamps, barrier-terminal connectors, and a custom PC board for smaller components.

You can buy a ready-made aluminum chassis for all these parts, or you can "roll your own" as suggested here. In either case, the unit attaches to the bottom-plate assembly leaving room for the card rack and battery.

Figure 8-6 is a schematic diagram of the circuitry to be included on the main power-panel chassis. Figure 8-7 is the schematic for the custom PC board that can be either mounted under the power panel or attached to the side of it. I originally planned to fasten this particular PC board to the underside surface of the power panel, but then I found it especially advantageous to have ready access to test points on that board, so I finally mounted it to the side of the panel.

The parts list for both the power-distribution and motor-control circuits and the associated PC board is shown in Table 8-2. The main motors for the Rodney unit are also specified on this list, although you might not choose to install them at this time. The LED indicator lamps on the power panel are adequate for testing the system up through the motor-control circuit.

There should be nothing surprising about the schematic for the power and motor-control chassis circuit. You have seen most of these components in one way or another in earlier figures of this chapter. Use the general panel layout in Fig. 8-8 as a guide for placing the components. Also see Fig. 8-9.

Diodes D301 through D304 swamp out the flyback current from the relay coils whenever their drive transistors are switched off. These diodes can be soldered between pins 2 and 7 on the 8-pin octal sockets provided with the relays.

The parts list also shows wires of several different types and gauges. The power-handling wires on this unit should be made of stranded wire of size 18 or larger. This wiring includes all of that connected to (+) BATC, the fuses and pin 5 of each of the relays, as well as the (-) BATC wiring to the COMM terminal and pin 6 on each relay. All of the remaining wiring, except that going to the LEDs and

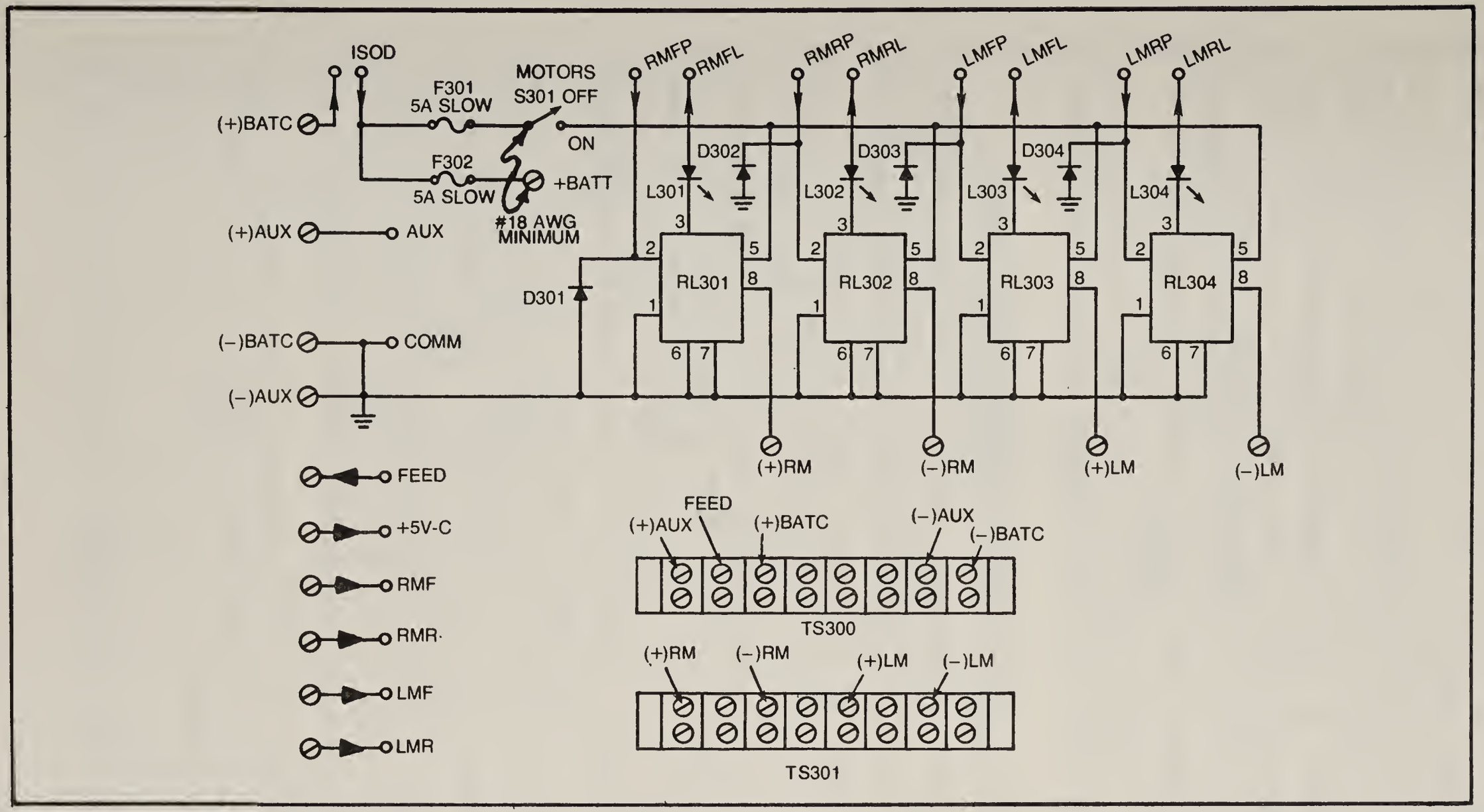


Fig. 8-6. Wiring diagram for the power panel.

pin 2 of each relay, should be the 22-Ga. solid hook-up wire. The wiring to pin 2 of the relays and LEDs can be the finer Kynar wire-wrap variety.

The printed-circuit board in Fig. 8-7 includes a feature that must be explained. Note from the diagram in Fig. 8-5 that two diodes (D1 and D2) are used to control the distribution of input power whenever the Rodney system is operating from both the battery charger and auxiliary power supply. These are relatively high-current diodes that appear as part of a bridge rectifier assembly on the circuit board in Fig. 8-7. This is a somewhat unusual application for a full-wave bridge rectifier package, but the scheme is a tidy one that works out very nicely.

A suggested drilling and bending plan for the power panel is shown in Fig. 8-10. Figure 8-11 shows the layout for the PC board.

INSTALLING AND TESTING THE MOTOR-CONTROL SYSTEM

After completing the wiring on the bottom of the power panel, bolt it down to the bottom plate of the mainframe unit, just ahead of the card-rack assembly. The indicator lamps should be situated near the outer edge of the bottom plate for easy viewing.

Doublecheck the interconnections between the power panel and PC board. For preliminary testing purposes, there should be no connections between the PC board and the I/O board at this time.

Begin your preliminary tests by connecting the +12V output of the auxiliary power supply to the (+) AUX terminal on the power panel, +5V from the power supply to the +5V-C terminal, and one of the COMM connections from the power supply to one of the power panel's COMM terminals.

You should not notice any responses at this point. Now, connect one end of a test jumper wire to a COMM terminal on the auxiliary power supply. Touch the other end to the RMF pin on the PC board. You should be able to see the RMF relay (RL301) pull in and the RMF indicator lamp should go on. If this does not happen, check to make sure the wiring is correct and complete between the PC board and power panel. Also doublecheck the appropriate connections on both units.

You can test each of the four motor-control relays in this fashion, making certain each one of them is energized whenever its corresponding motor-control input terminal is grounded.

Once you are sure these circuits are operating as they should be, it is time to wire the four motor control inputs on the PC board (RMF, RMR, LMF and LMR) to the pins of the same designation on

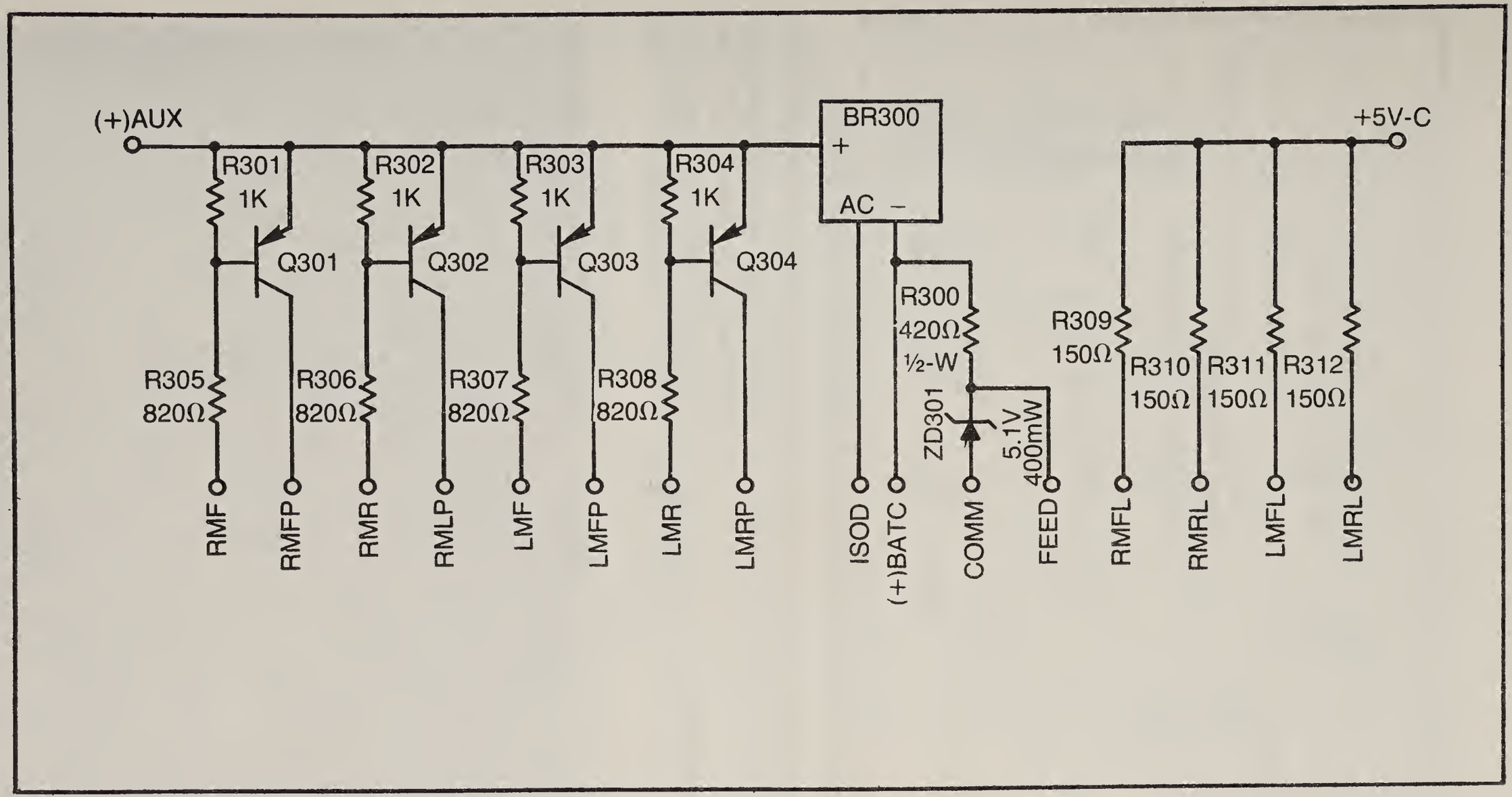


Fig. 8-7. Schematic diagram of the printed-circuit board in the motor-control circuitry.

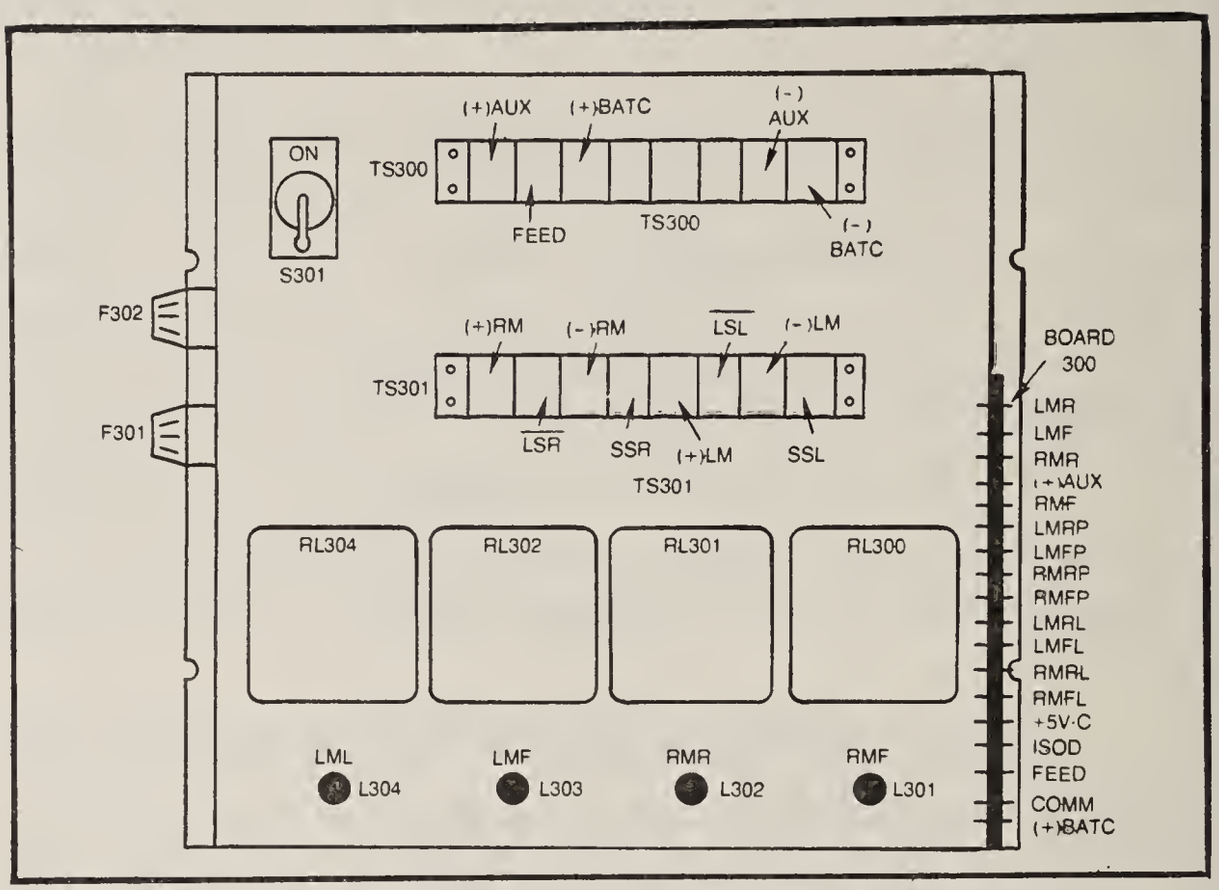


Fig. 8-8. Mechanical arrangement of components on the power panel.

the 100-pin connector of the I/O board. These connections and all others running between the I/O board and motor-control assembly ought to be made through a 12-pin plug-and-cable assembly. I wired them directly in my prototype model, however. Frankly, I was quite anxious to see the motor-control system operating from the main I/O bus.

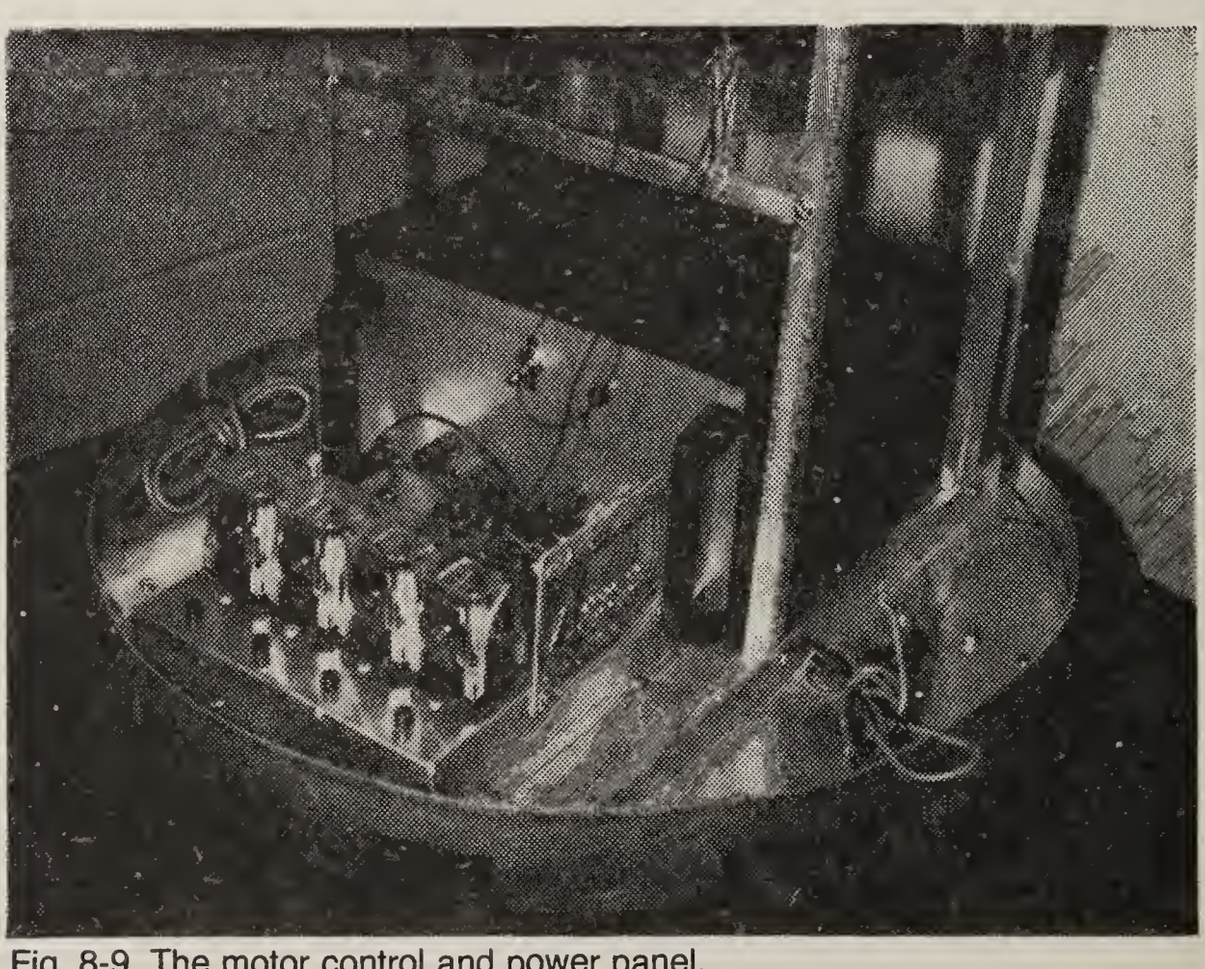


Fig. 8-9. The motor control and power panel.

Table 8-3 summarizes the wiring between the I/O board and motor-control circuits. The exact wiring for the motor sensing LEDs and phototransistors has not yet been described, although the wires are specified here. You can either put off this part of the job until tests on the motor-control section are complete, or you can peek ahead in this chapter to see what is going on and wire them in now.

With full power from the auxiliary power supply still going to the motor-control system, apply 12V power to the CPU and I/O boards. You should now be able to control the action of the relays and motor-status indicator lamps from the system's front panel. To access this control circuit from the front panel, set the RUN/PROGRAM switch to PROGRAM and set the address switches for Port-0 tests—8OOH. Now, you should have complete control over the relays from data switches D0 through D3. Of course, you have to depress the LOAD pushbutton momentarily to load any sequence of logic conditions.

LMR should be responsive to the setting of D0, LMF to the setting of D1, RMR to the setting of D2, and RMF to the setting of D3. If this combination is scrambled for any reason, you must change the four corresponding connections between the I/O board and PC board at the motor-control unit.

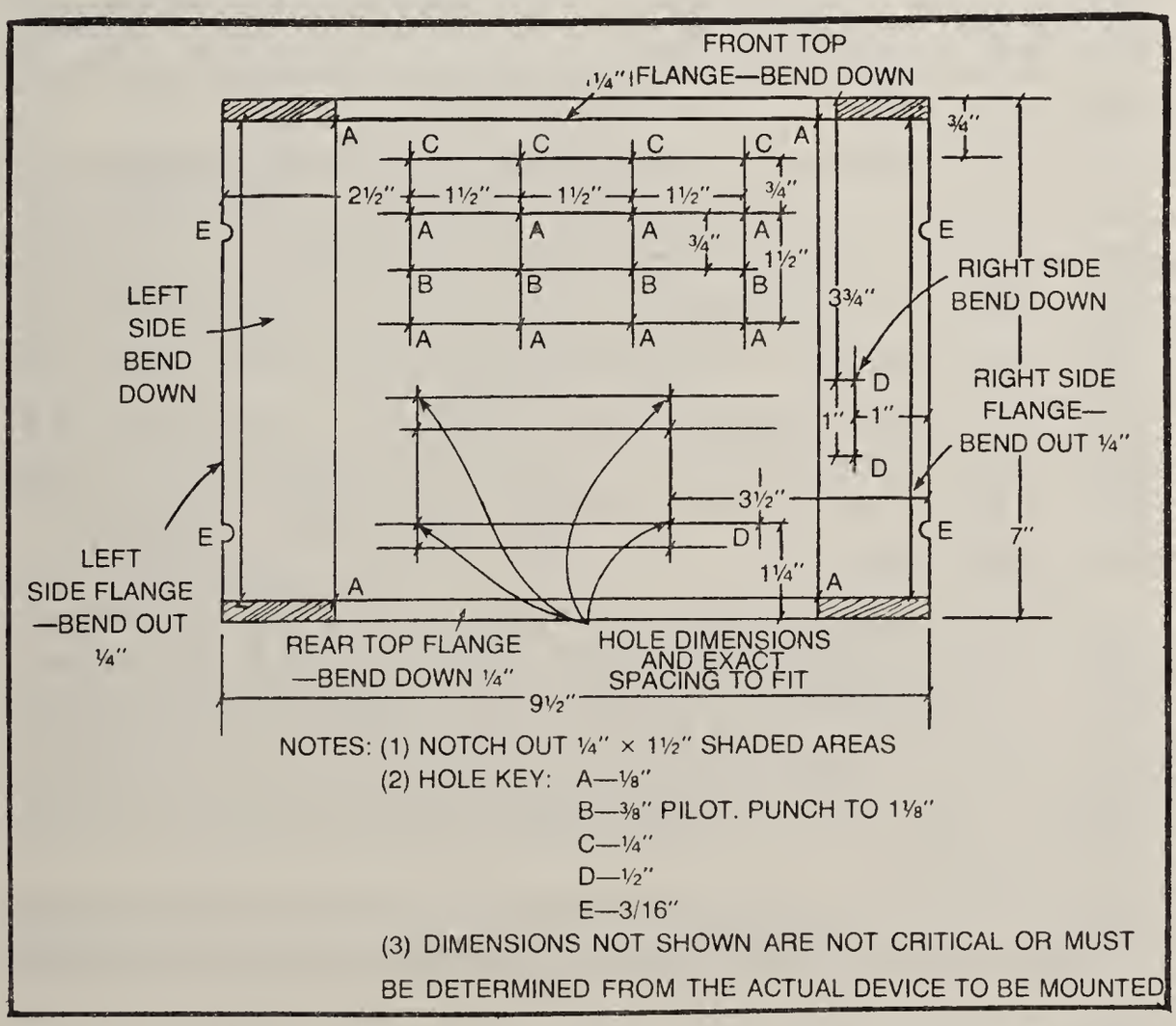


Fig. 8-10. Suggested bending and drilling pattern for the power panel.

Table 8-3. Summary of connections between the I/O board (Board 200) and the power-distribution and motor-control assemblies.

Function	I/O Board Pin No.	Power/Control Connection
$\overline{\text{LMR}}$	25	PC Board
$\overline{\text{LMF}}$	24	PC Board
$\overline{\text{RMR}}$	27	PC Board
RMF	26	PC Board
+5V-C	Z	PC Board
$\overline{\text{LSR}}$	f	TS301
LSL	h	TS301
SSR	c	TS301
SSL	d	TS301
+12V	1	(+)AUX TS 300
COMM	50	(−)AUX TS300
FEED	28	FEED PC BOARD

INSTALLING AND TESTING THE MOTOR SPEED SENSE CIRCUIT

Figure 8-12 is the wiring diagram for the motor-motion sensing phototransistors and LEDs. The theory of how this circuit operates has already been described in connection with Fig. 8-4, so there's little more to say than "hook it up."

To test the circuitry, apply 12V power from the auxiliary power supply, switch off the motors (if you have them wired into the system), and set the system for the PROGRAM mode of operation. Address Port 0 operations with hex code 800H and then LOAD data F0H. All four motor status lamps should go off and the data lamps should read F0H.

Next, turn on the FSLL motion-sensing LED by LOADing data E0H. D4 of the data display indicates the status of the left-motor phototransistor, but since we do not know whether or not that motor is in a position to let LED light fall onto the phototransistor, there is no way to tell whether or not D4 should be on. Therefore, rotate the left motor by hand, keeping a close eye on the D4 data lamp. If it blinks on and off as you turn the wheel, everything is in good working order. If things aren't working out this way, you have to perform a couple more tests.

Connect a logic probe or voltmeter to the $\overline{\text{LSL}}$ terminal on TS301 and alternately LOAD data F0H and E0H. If the turn-on signal is reaching the LED, the logic level should be 0 when loading E0H and logic 1 when loading F0H. If you aren't getting this kind of response, check the circuitry between the system data bus and that LED connection.

If, on the other hand, your tests show you are indeed getting the logic-0 turn-on signal at the LSL terminal, the problem is either

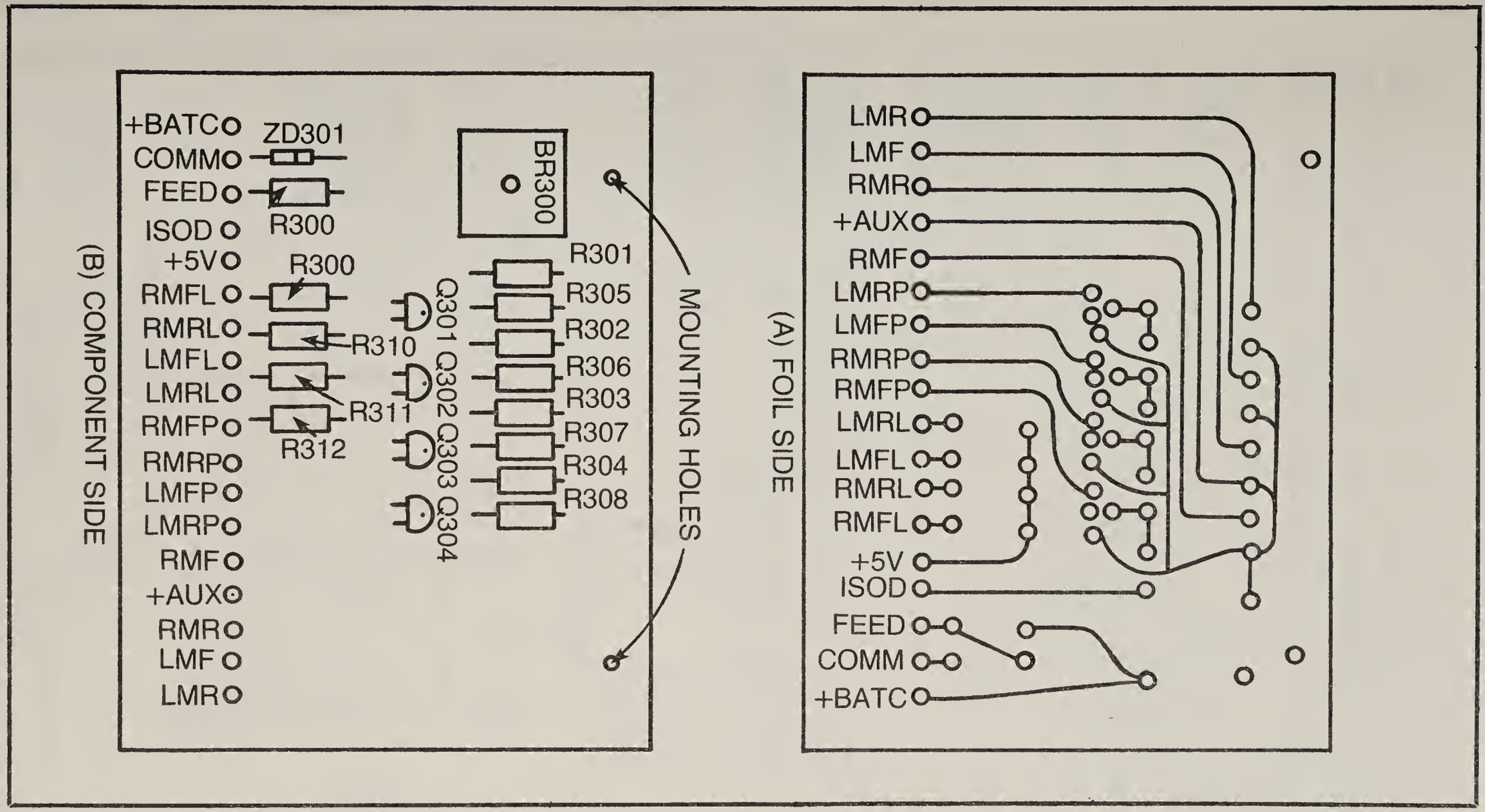


Fig. 8-11. Layout of the PC board. (A) Foil side. (B) Component side.

the LED itself is improperly wired or something is wrong with the associated phototransistor circuit. Check out the phototransistor circuit for possible wiring problems, and, of course, don't overlook the possibility that the two motion-sensing circuits might be reversed.

Check the right motor-sensing circuit in a similar fashion, LOADing FDH and observing the response of data lamp D5, the right-motor motion-sensing lamp.

As a final test, LOAD data C0H and manually turn the motors, either one at a time or at the same time. Data lamps D4 and D5 should blink on and off as their respective motors turn.

If you haven't already done so, you can now connect the motors to their designated terminals on TS301: the positive and negative leads of the right motor to (+)RM and (-)RM, and the left-motor leads to (+)LM and (-)LM. Remember to solder 0.1μF capacitors to the motors' power terminals as shown in Fig. 8-3.

Check out the system by leaving it in the PROGRAM mode and addressing 800H. Do not enter a data word that calls for running both motors at the same time—the auxiliary power supply cannot handle the current drain for *both* motors.

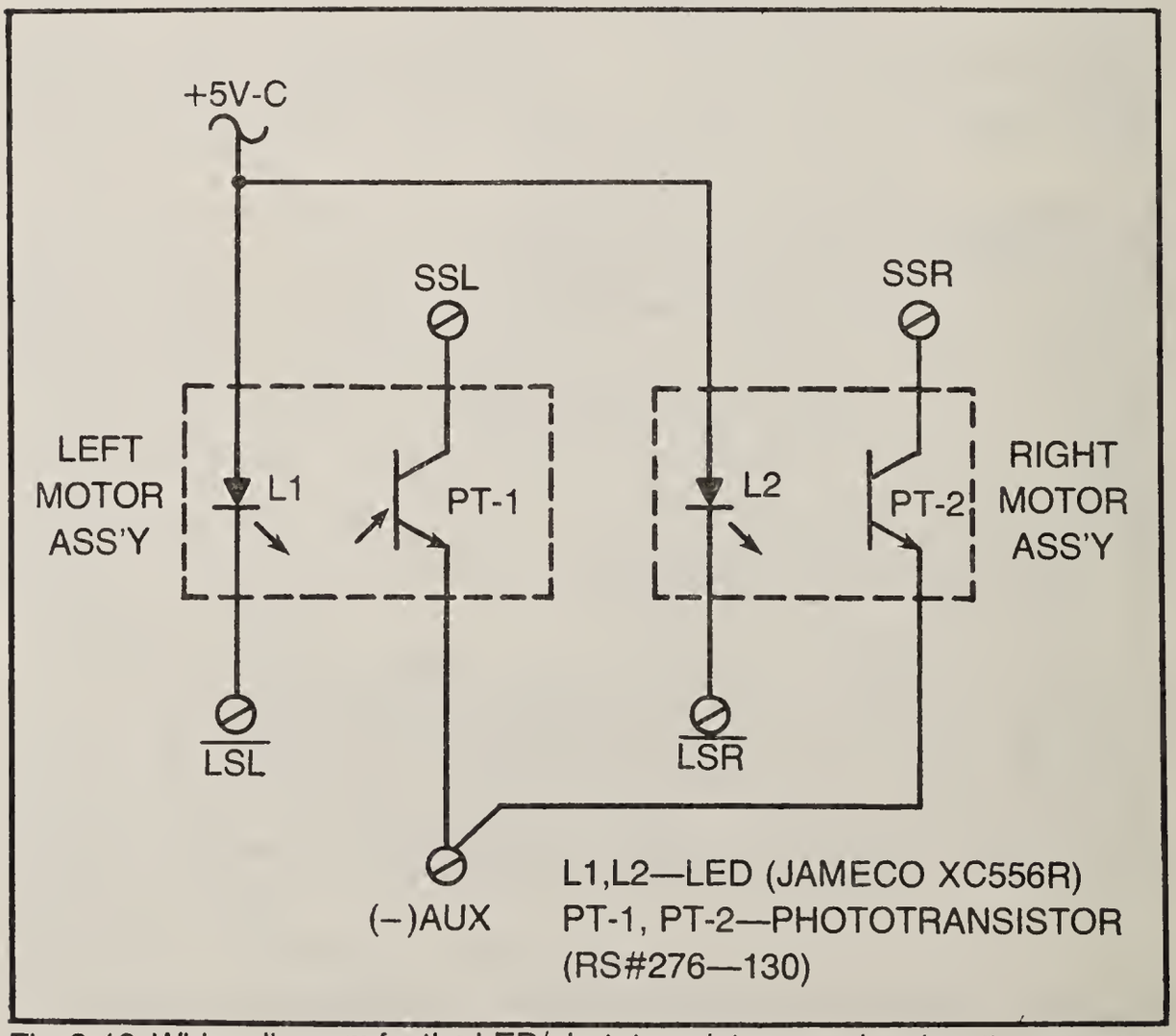


Fig. 8-12. Wiring diagram for the LED/phototransistor speed and motion-sensing system.

Check the left motor and its motion-sensing circuit by LOADing E1H and then E2H. In the first case, the left motor should run in reverse, and D4 should blink on and off. In the second case, the motor should run in a forward direction, and D4 should blink. If the motor does not run in the directions indicated here, simply reverse the connections to (+)LM and (-)LM on TS301. Of course, the motor status lamps should indicate an LMR in the first case and LMF in the second case.

You can check out the right motor system by LOADing data E4H and E8H. In the first case, the right motor should turn in a reverse direction, the RMR motor status lamp should go on, and D5 should blink on and off. In the second case, the right motor should turn in a forward direction, the RMF motor status lamp should go on, and D5 should blink.

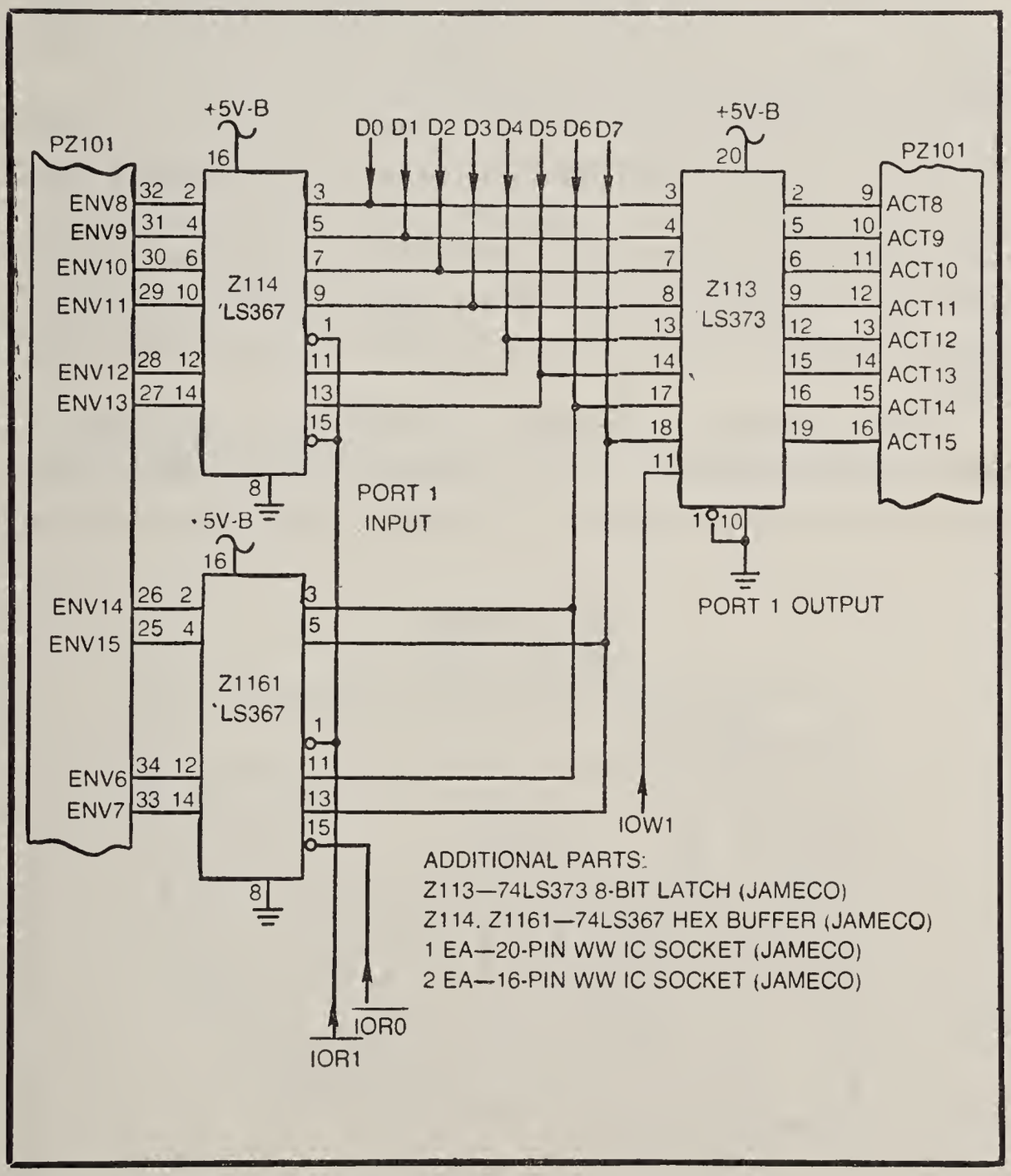


Fig. 8-13. Additional I/O buffer circuitry on the CPU board.

ADDING THE CIRCUITRY FOR "FEED" SENSING

The Alpha-Rodney program described in Chapter 1 cannot be run unless the machine is capable of responding in a positive fashion to contact with the battery charger. The actual feed sensing point is across the zener diode shown in Fig. 8-5. Whenever the positive terminal of the battery charger is connected to (+) BATTERY CHARGER (and of course a good COMM connection is also made), the voltage at the cathode of ZD301 rises to about +5.1V.

This FEED point must be interfaced with the system as ENV6—D6 of ENVL. Figure 8-13 shows additional I/O buffers to be added to the CPU board, including the connections for FEED. Fig. 8-14 then shows the suggested layout for these additional ICs.

The FEED signal reaches the CPU board, however, through the 40-pin cable running to the I/O board, so that means you must add the simple circuit in Figure 8-15 to the I/O board.

As an overall view of the FEED signal, it starts at ZD301 on the PC board in the power-distribution circuitry, passes through a wire to pin 28 on the I/O board edge-card connector, and then goes across a 470-ohm resistor to pin 34 of PZ201. After that, the signal is picked up at the other end of the 40-pin connector (pin 34 of PZ101) where it becomes ENV6 at pin 12 of Z1161—a Port 0 *read* input buffer.

After connecting this circuitry, the FEED status will appear at data lamp D6 whenever Port 0 is addressed in the PROGRAM

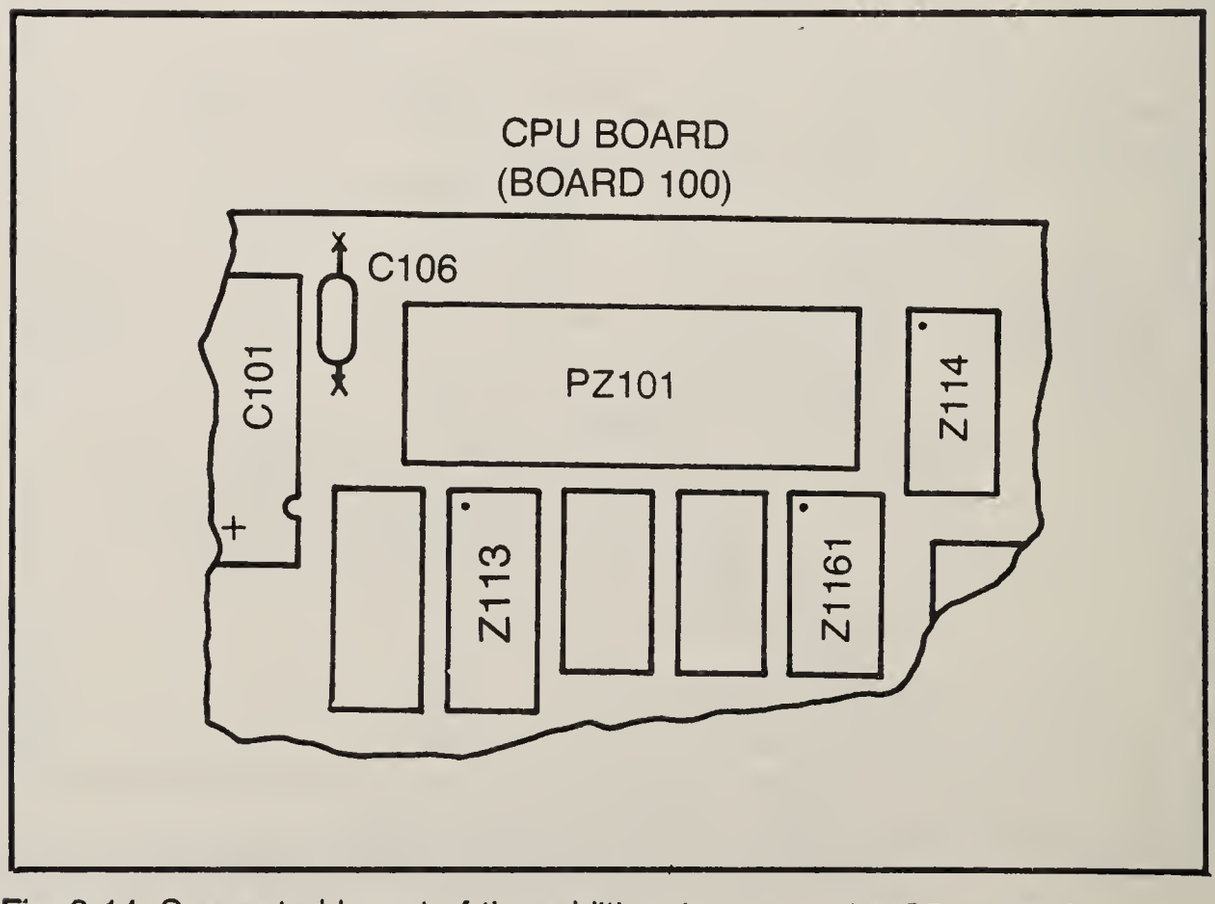


Fig. 8-14. Suggested layout of the additional parts on the CPU board.

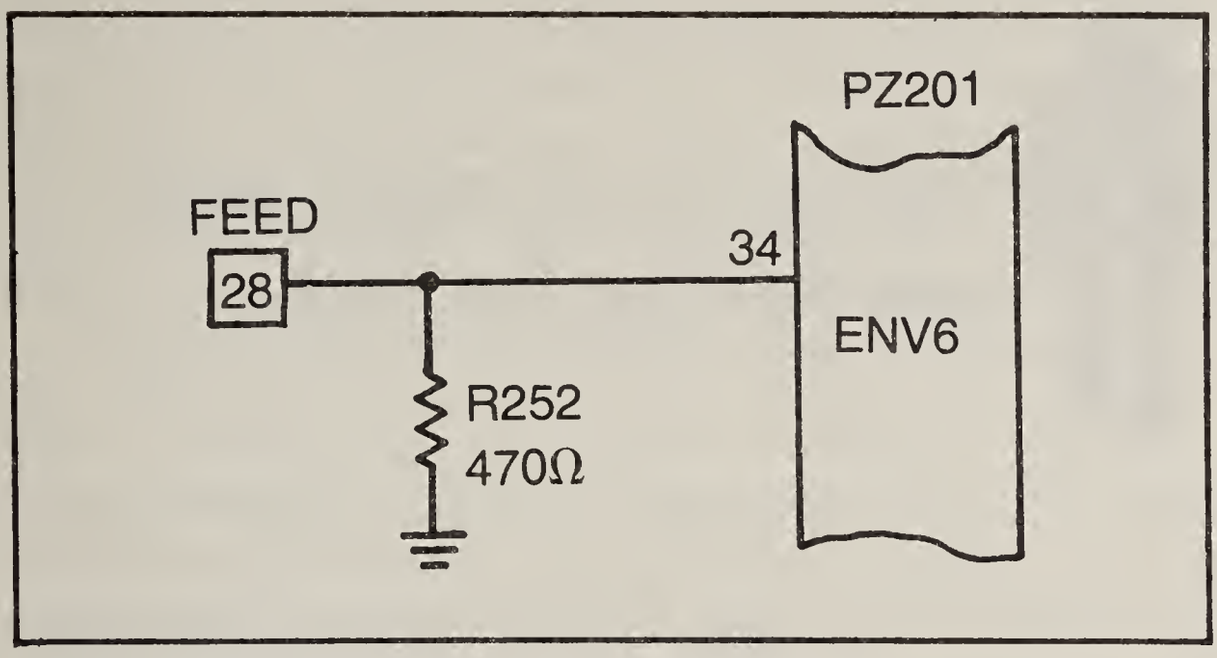


Fig. 8-15. FEED wiring on the I/O board (Board 200).

mode. Under these circumstances, the lamp should go off unless +12V is applied to the (+)BATC connection on the power-distribution panel.

9 Adding the Microprocessor and Building the Nest

Whatever your source of good fortune and success might be, this is the time to call upon it. You'll need a lot of positive thinking and all the technical talent you can muster. The work in this chapter represents the worst and the best of everything you do with Rodney. You are going to have to dig in and make things work on your own, because the work is tricky.

This is the point where all the circuitry you have built thus far is transformed from switch-operated DC control to r-f control. A clocking frequency of 2.5MHz is r-f in anyone's book, and this is what the microprocessor introduces to your circuitry. All the wiring running between ICs on the CPU board and all that wiring running between the CPU and I/O boards becomes an inductive and capacitive nightmare now.

I am sure there are some experimenters who haven't had much experience in this area and are wondering what the fuss is all about. Maybe you have wired everything just as I described so far in this book, and since my model works, doesn't it seem logical yours should fire up and run right? The answer is *no*, not necessarily, because I didn't say how long to cut each wire and exactly how to route it from one point to another.

Your circuit is going to be different from mine and everyone else's. The wiring diagrams are the same, but the stray capacitance and inductance are bound to play different roles in each case.

I am dealing with this bad news now in order to get you prepared for some potential problems later on. There *are* solutions to the problems, and I'll guide you along the way. You are going to have to start adding decoupling capacitors all over the CPU board and maybe on the I/O board, too. I don't know how many you will need, so I won't bother numbering them, or even showing them, on the schematics. You will be adding these capacitors until the microprocessor system works properly and reliably. That might take one evening or a week.

That's the bad news, and there will be more to do with it later. Now for the good news. Adding the microprocessor and its immediate buffer circuits gets your Rodney project running on its own for the first time. And, when that happy moment arrives, you will have paid your dues and have every right to run around the house shouting "eureka" and things such as that. I've played around with a lot of neat circuits over the years, but nothing parallels the moment Rodney sprang to life under the control of the microprocessor.

You should build up the nest assembly now, too. There is a little bit of tricky mechanical work involved in building the nest and Rodney's charging rings, but compared to ironing out the microprocessor despiking problem, building the nest seems like relaxation.

THE MICROPROCESSOR AND BUFFER CIRCUITS

Figure 9-1 shows the 8085 microprocessor chip and the components immediately associated with it. A few definitions and special explanations are in order.

SID—This is the serial data input. You won't be using it for a while, but it will eventually serve as the input point for data coming from a cassette tape player.

RESET IN—Pulling this point down to logic 0 immediately stops all microprocessor operations, and upon returning the point to logic 1, the microprocessor resumes operation taking its first instruction from the lowest address in the program RAM. A time-constant circuit (C105 and R100) pulls this point down to logic 0 whenever DC power is first applied to the system, giving Rodney a "power-on clear" feature.

RESET IN is also connected directly to the RESET switch on the front-panel assembly. You can stop microprocessor operations at any time by switching on the RESET switch, and when it is switched off, microprocessor operations begin from address 000H.

X1 AND X2—These are the connections for the system's 5 MHz crystal, X100. This frequency is internally reduced to 2.5 MHz, thereby establishing a constant 2.5MHz clocking frequency for all microprocessor operations.

AD0 THROUGH AD7—These eight lines carry a multiplexed combination of addresses A0 through A7, and data D0 through D7. The different kinds of information are demultiplexed (or *sorted out*) by the three buffer circuits in Fig. 9-2.

A8′ THROUGH A11′—These are the higher-order address lines which are buffered by Z103 in Fig. 6-3. Since these lines are not

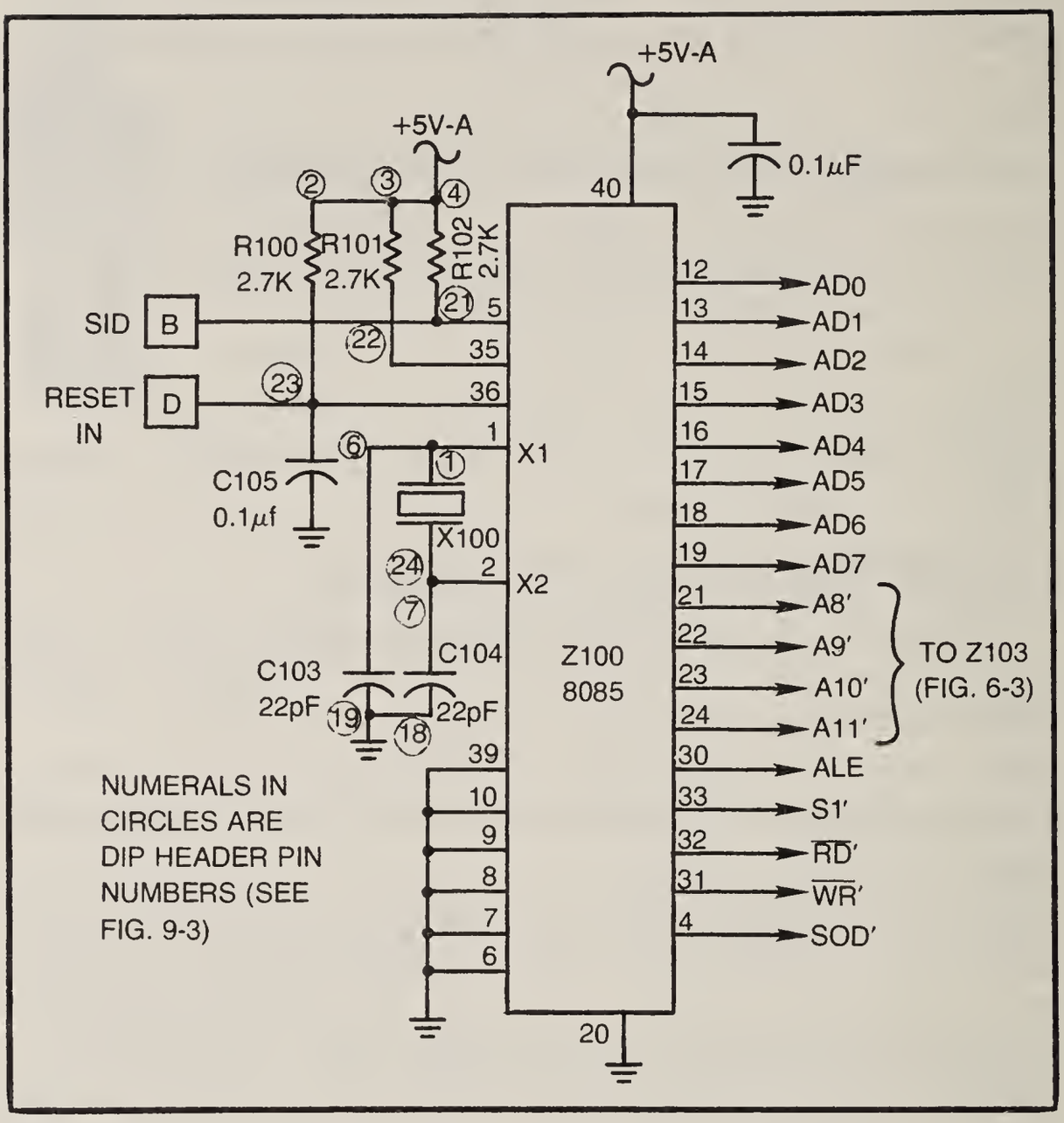


Fig. 9-1. The microprocessor circuit.

multiplexed with anything else, there is no need for demultiplexing them.

ALE—This one-line signal is responsible for signalling the difference between address and data information on lines AD0 through AD7. Whenever ALE is at logic 1, it informs the outboard demultiplexing scheme that address information is present on those lines. Otherwise, the information is data or not relevant.

S1′—This is a status signal that informs outboard circuitry of the microprocessor *read* or *write* status. In essence, it is a R/$\overline{W}$ signal, indicating *read* operations when at logic 1 and *write* operations when it is at logic 0.

$\overline{RD}$ and $\overline{WR}$—These are active-low pulses that indicate exactly when the microprocessor is expecting valid input data on the data bus (RD=0) or is generating valid output data on the data bus (WR=0).

SOD'—This is the serial data output connection which will be used only for transcribing programs and memory information onto cassette tape.

Figure 9-2 shows the microprocessor buffer circuits which, incidentally, also demultiplex the AD lines. Z102 is an 8-bit data latch which is enabled only when the system is in the RUN mode. Otherwise, its outputs, A0 through A7, go to a tri-state condition.

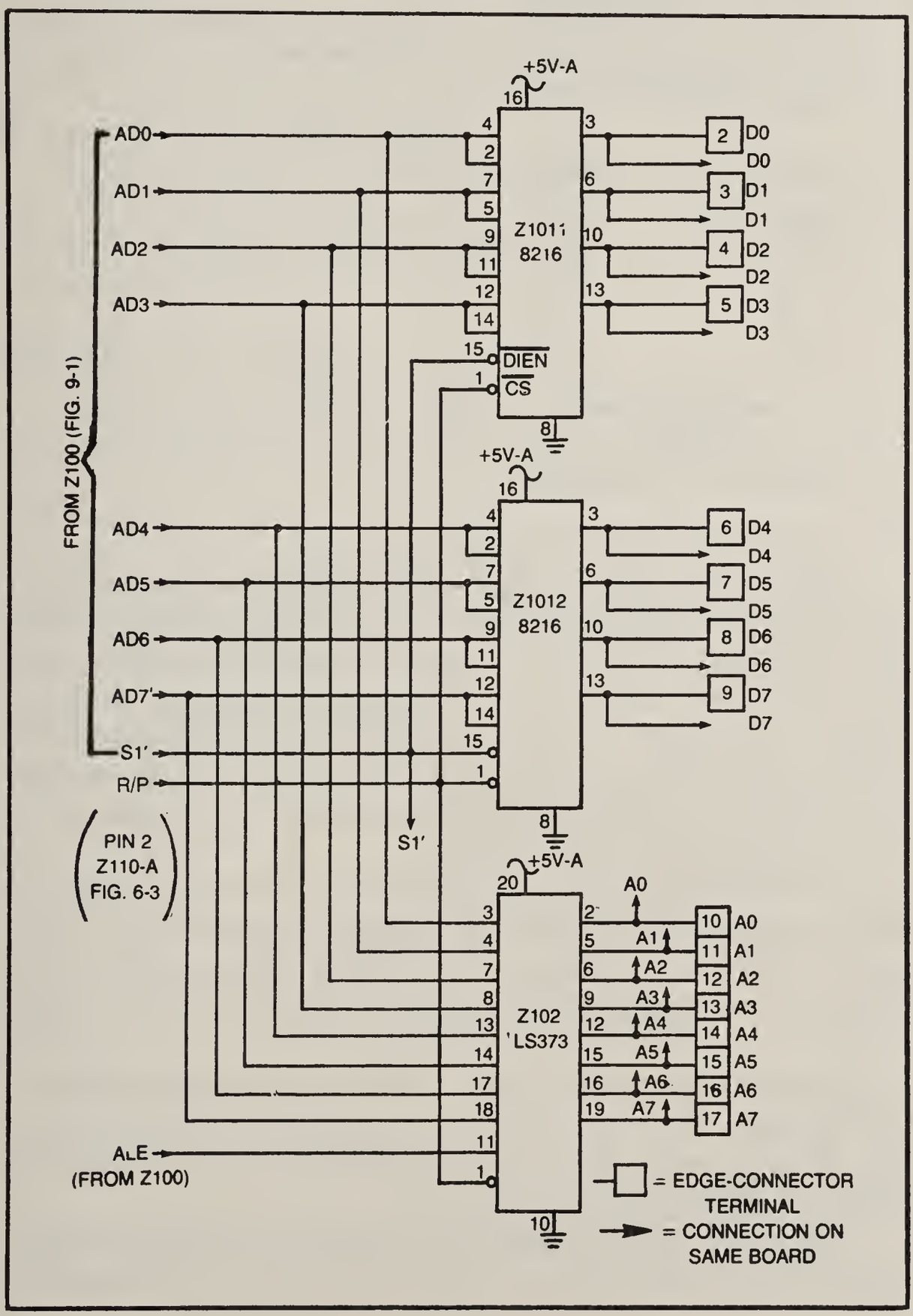


Fig. 9-2. The microprocessor buffer circuit.

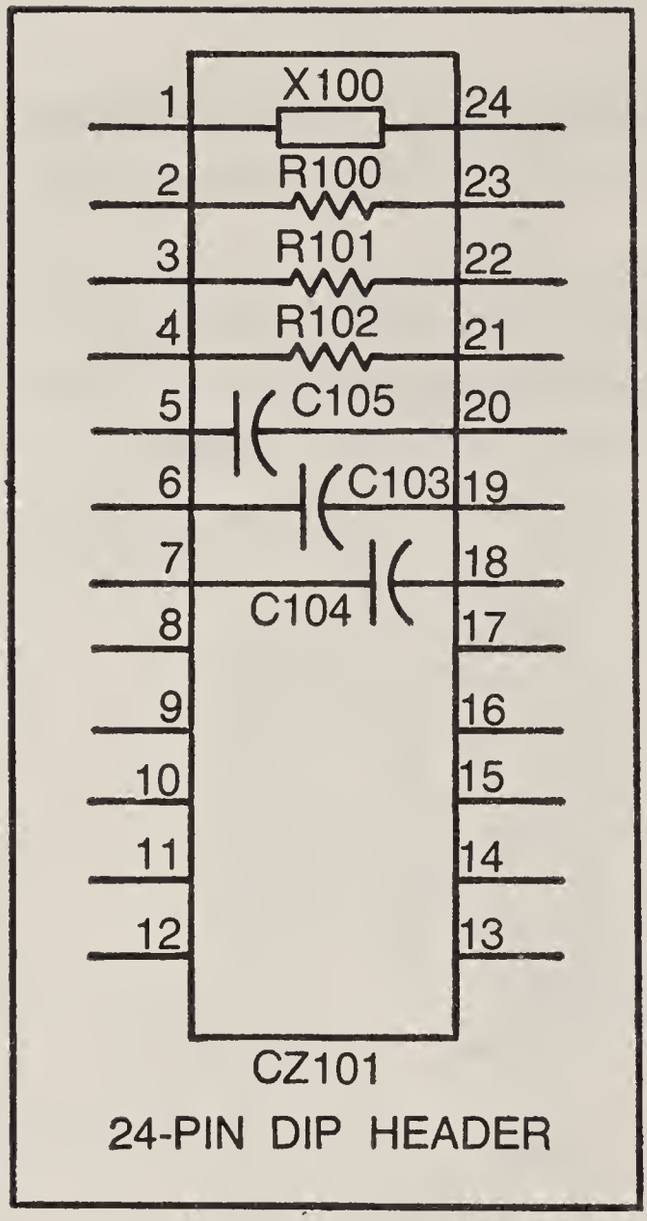


Fig. 9-3. Dip header for the small components.

Whenever the system is in the RUN mode, new information is latched and held valid at the outputs whenever ALE shows a positive pulse. The inputs to Z102 are connected to the eight AD lines, and since ALE occurs only when valid address information is on these lines, it follows that Z102 picks up and latches the lower-address byte for the system. Connections A0 through A7, in other words, carry only valid address information, data from the AD lines never reach this part of the address bus.

The microprocessor must be able to send out a byte of data as well as receive it from the system data bus. Therefore, lines AD0 through AD7 are also connected to a pair of bus transceivers (bi-directional bus drivers). The devices in this case are Z1011 and Z1012.

Like the address latch, these bus transceivers are enabled by the $\overline{R}$/P signal. This means they are operative only when the system is in the RUN mode. Otherwise, the microprocessor is completely isolated from the data bus.

Whenever the bus transceivers are enabled in the RUN mode, S1 determines the direction the data flows. When S1′ is at logic 0,

data flows from the AD outputs of the microprocessor to the D0 through D7 lines of the system data bus. When S1′ goes to logic 1 (indicating a *read* operation), data flows from the data bus to the AD connections of the microprocessor. Address information is sometimes present on the data bus, but the $\overline{RD}$ and $\overline{WR}$ pulses make certain no I/Os are connected to the data bus at that time.

INSTALLING THE BUFFERS AND MICROPROCESSOR

Install the microprocessor and buffer circuits shown in Figs. 9-1 and 9-2. The small components associated with the microprocessor chip can be mounted on a DIP header as indicated in Fig. 9-3. The numerals inscribed in circles in Fig. 9-1 correspond to the pin numbers on this header circuit.

Figure 9-4 is the recommended layout for these components on the CPU board. The parts list is in Table 9-1. When you have completed the wiring, install the buffer ICs, but do not install the microprocessor chip yet.

Static Tests for the Buffers

With the three buffer ICs installed (Z1011, Z1012 and Z102), apply 12-V power to the system and set things up for the RUN mode

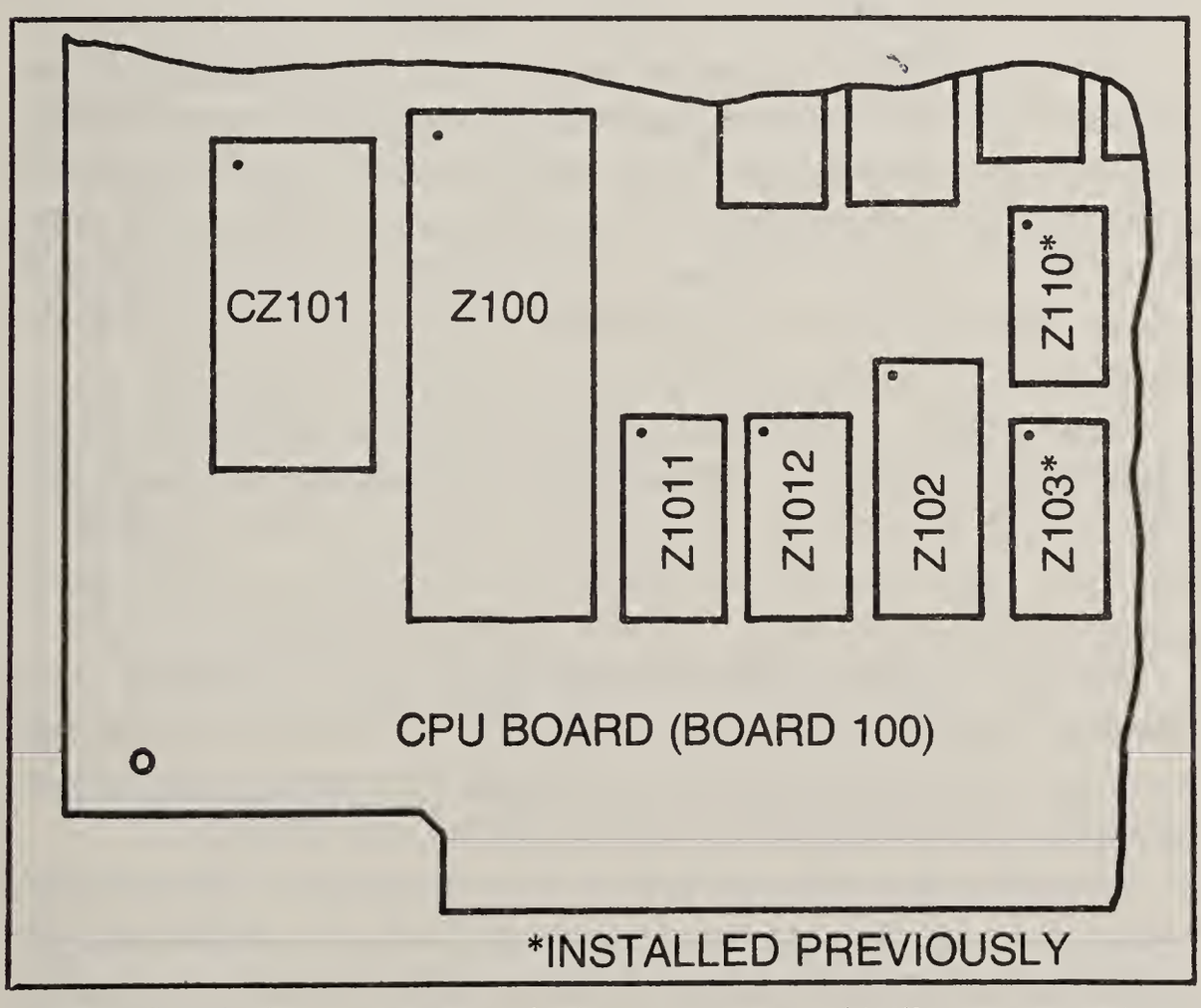


Fig. 9-4. Physical arrangements of microprocessor and buffer components on the CPU board.

of operation. The socket for Z100 should be empty, making it a good place to insert test wires. Cut a couple 6-inch lengths of Kynar wire-wrap wire and strip about one-half inch from each end. These are going to be jumper wires for the buffer tests.

First check the data buffers' ability to pass data from the AD lines to the data bus. To start, connect one of the special test wires between COMM (pin 20) and S1′ (pin 33) of the microprocessor socket. Insert the ends of the wire right into the places where the pins belong.

Then, connect one end of the second test wire into the Vcc (pin 40) slot for the microprocessor. Touch the opposite end of that wire to AD0, AD1, AD2, and so on. As you do this, use a voltmeter or logic probe to check the logic levels at the data bus connections on the edge-card pins of the CPU board. When you connect the jumper to AD0, you should see a logic-1 level at D0, pin 2 on the 100-pin card connector. In the same way, when you connect the jumper to AD1, you should see logic-1 at D1, and so on. Check the system through AD7 in this way. Then, connect one end of the second wire to COMM instead of Vcc and run through the sequence again, looking for logic 0 at the data bus connections. This completes the static checks for microprocessor-to-data bus transfers.

Now, you have to test the same circuit for data flow in the opposite direction. Remove the test wire running between COMM and S1′ and reconnect it so it is running between Vcc (pin 40) and S1′ of the microprocessor. This time you will need an ordinary clip-type jumper wire to apply COMM and Vcc to the data bus connections and monitor the logic levels at AD0, AD1, AD2, etc. at the microprocessor.

Connect one end of the clip-type jumper to a +5V source (the output of the auxiliary power supply works nicely) and touch the other end to D0 (pin 2) on the CPU board's 100-pin connector. Check the logic level at the AD0 (pin 12) connection of the microprocessor socket. It should show a solid logic-1 level.

Check all eight data bus and AD connections in this way, then wire the test jumper to COMM instead of +5V and repeat the sequence looking for logic-0 levels this time. This completes the test routine for the bi-directional buffers, A1011 and Z1012.

Checking the address buffers is a bit simpler. Remove all jumper wires and connect one of the special test wires between ALE (pin 30) on the microprocessor socket and +5V. Connect one end of the second test wire to +5V as well, and touch the other end to the microprocessor's address terminals, AD0 through AD7 and then A8′ through A11′. With each test, check the logic level at the corre-

Table 9-1. List of parts to be added to the CPU board.

Semiconductors

Z100	8085 8-bit microprocessor	(Intel)
Z1011, Z1012	8216 quad bus transceiver	(Jameco)
Z102	74LS373 octal latch	(Jameco)

Resistors and Capacitors

3 ea. 2.7k, ¼W resistor
2 ea. 22pF ceramic capacitor (Jameco)
2 or more 0.1μF mylar capacitor—see text for explanation (Jameco)

Wire-Wrap IC Sockets

1 ea. 40-pin WW socket (Jameco)
1 ea. 20-pin WW socket (Jameco)
2 ea. 16-pin WW socket (Jameco)
1 ea. 24-pin WW socket (Jameco)

Other Components

1 ea. 5 MHz crystal (Jameco CY7A)
1 ea. 24-pin DIP header (Jameco)

sponding address-bus pin on the 100-pin edge connector, pins 10 through 21, for a logic-1 level. When connecting AD0 from the microprocessor socket to +5V, for instance, you should see logic 1 at pin 10 on the 100-pin connector.

You can complete this series of static tests by connecting the special test wire between COMM and the microprocessor's address terminals instead of between +5V and those terminals. Check for logic 0's at the corresponding address-bus pins on the 100-pin connector.

I realize this is a time-consuming and tedious series of static tests, but they will point out any wiring problems in the microprocessor bus system. It is important to uncover any problems at this time so that the dynamic microprocessor tests won't be so confusing.

Firing Up The Microprocessor

Carefully check the wiring of the control lines from the microprocessor socket, $\overline{RD}$, $\overline{WR}$, SOD', etc. You might like to use an ordinary ohmmeter continuity test in these cases.

TURN OFF ALL POWER TO THE SYSTEM, and carefully slip the 8085 microprocessor into place. Have you installed the 24-pin DIP header for the small components? If not, do it now.

Is the RESET switch on the front-panel assembly connected to RESET IN of the microprocessor? Pin D of the CPU board should be going to pin D of the I/O board and from there to the RESET switch via one of the 16-pin plug assemblies. (See Fig. 5-9.)

Set the system to its PROGRAM mode and turn on the RESET switch. Apply power to the system. Nothing of any real consequence should happen because the microprocessor is isolated from the system by the PROGRAM mode and disabled by the RESET function.

If you have access to a fairly decent oscilloscope, you should be able to see a 5MHz waveform at pin 1 of the microprocessor and a 2.5MHz waveform at pin 37 (a CLK output function that isn't used in the Rodney system). All of the address and data lines *at the microprocessor* should be "floating" at about 2.5V.

Hold on —here comes the real test of everything you've done so far! You are going to enter a simple program into the program RAM and see how well the microprocessor executes it. If you are going to have the problems outlined in the opening paragraphs of this chapter, you're going to run into them now.

Set up the panel address for 000H and LOAD 00H. This is a NOP (no operation) command. Then set the address to 001H and LOAD data C7 (a command that tells the system to return to address 000H). You are telling the microprocessor to do nothing and then return to doing nothing again. Ideally, the system will buzz back and forth between these two commands at an r-f rate.

In a semi-assembly programming format used for most of the work remaining in this book, this simple program will look something like this:

```
000  00        TEST:
001  C7        ;RETURN TO TEST
```

Now set the RUN/PROGRAM switch to RUN. If the RESET switch is still ON (as it should be at this point), nothing should happen. Return the system to PROGRAM after a few seconds and check the content of the program RAM. The data at 000H address should still be 00H and the data at 001H should still be C7H. If it is not, something is wrong with the RESET wiring between the front panel and the microprocessor. Do not go any further with the testing until you are convinced the program RAM holds the 2-step program when switching the system between RUN and PROGRAM.

Assuming things are holding together so far, set the system for RUN and turn off the RESET switch. Wait for a few seconds, set the RESET switch to OFF, and return the system to PROGRAM. Is data 00H still at address 000H and is data C7H still at address 001H? Probably not. If it is, you are exceedingly lucky!

Chances are quite good that this test has scrambled the simple 2-step program. Why? With inadequate capacitive decoupling on the CPU board, the microprocessor picks up some strange and meaning-

less commands that eventually replace your program with a "garbage" program of some sort.

The motor-control circuit probably went crazy for a moment anyway. That's because some of the "garbage" commands addressed motor-control functions. After the 2-step program gets scrambled, the microprocessor flies off into strange places in the program RAM that carry meaningless instructions and data.

It's a mess, isn't it? Maybe it is of some comfort to know that someone else recognizes your predicament. Try reloading the 2-step program and running it several times, always going to RUN before turning off the RESET switch and turning on the RESET switch before going back to PROGRAM.

Things will probably get scrambled every time. If the program isn't scrambled by some quirk of good fortune, try it again for several minutes. You cannot afford to hedge on this bet—programs must remain intact in the program RAM indefinitely.

Let's suppose the program isn't getting scrambled and you are wondering what this fuss is all about. In this case, the system just might not be working right anyhow. Try this program:

```
000   21    00 08 OTST:    LXI     H,PORTO
003   3E    FF             MVI     A,FFH
005   77                   MOV     M,A
006   C7                   RST     0
```

If you have been doing your homework on 8085 instruction sets and programming, you will see that this program loads all 1's at the motor-control outputs (Port 0). The 3-digit numbers are address locations and the 2-digit numbers are hex data to be entered at the outputs. Since the first line of the program uses three different bytes of data, it figures they will occupy address locations 000H, 001H and 002H. That's why the second line begins with address 003H.

Load this little program but before RUNing it, turn off the motor status lamps by addressing Port 0 (800H) and LOADing data 00H. Then start the program by first setting the system to RUN and then turning off the RESET switch.

If the system is reading the program properly, the motor status lamps should turn on and remain on. They probably won't stay on, but they should.

If your system wasn't scrambling the 2-step TEST program, but the OTEST program doesn't run right, you know the system isn't executing *any* commands. If that's the case, you have to doublecheck a lot of circuitry, beginning with the program RAM tests in Chapter 7 and working all the way through the static tests in the previous section of this chapter.

Maybe it isn't so bad to have the 2-step TEST program fouling up after all. At least you know the microprocessor is executing commands. The commands are "garbage," but at least that part of the system is working.

The odds against the system running both of these dynamic test programs are tremendous.

SOLVING THE NOISE PROBLEM

Suppose you have arrived at the point where the 2-step TEST program gets scrambled every time you try it for more than a second or two. As long as this is happening, there is no chance at all that the system will run the OTEST program properly, so we can forget about that for the time being.

Getting the elementary TEST program running properly is both a matter of hard work and luck. Basically, you have to find the IC devices that are generating r-f spikes on the power supply which is fouling the program sequence. This means adding despiking or decoupling capacitors between the +5V and COMM connections of the devices causing the problem. The catch to the whole matter is that there is no way to determine which devices are at fault, and that can mean adding decoupling capacitors, virtually at random, until the system runs TEST without scrambling it.

Of course, it is possible to stick a decoupling capacitor on every IC on the CPU board from the outset, but that's a lot of capacitors. You are certainly welcome to take that approach if you want.

Otherwise, try attaching 0.luF capacitors *directly* across the Vcc (+5V supply) and COMM of the two program RAM chips, pins 18 and 9 of Z106 and Z017. Then decouple the microprocessor buffer ICs (Z1011, Z1012, Z102, and Z103) in the same fashion.

Load the 2-step TEST program and RUN it. If the program remains intact after a few moments of running, you are on the right track. Otherwise, you must add some more decoupling capacitors.

This time, add them to the function and I/O select ICs: Z108, Z109, Z110, Z105, Z111, and Z112. Load the TEST program and run it.

Bear in mind that the decoupling capacitors must be connected *directly* between the supply-voltage pins of the ICs they service. You won't make any progress at all by using indirect connections through pieces of wire-wrap wire.

Add some capacitors, make certain the RESET switch is ON, set the system for PROGRAM, enter the TEST program, set the system to RUN and then switch RESET to OFF. Wait several

moments, then turn ON the RESET switch, go to PROGRAM and look at the data for the program. Continue this procedure until the little program remains intact for an indefinitely long RUN time.

Who knows? Maybe you will end up putting a decoupling capacitor on every IC on the CPU board. If the TEST program still gets garbled, you have to start working on the I/O board as well.

I'm sorry. I wish you didn't have to go through this. It can be frustrating. Some competent manufacturer might one day produce a Rodney-type CPU board that has the decoupling problem solved for us. However, until such a time arrives, we have to live with the problem, just as any microprocessor engineer does.

If you find yourself losing patience with this part of the work, you might like to skip ahead in this chapter to the nest-building part. That will give you some relief from the tedious work.

Rest assured, the board will work once it is properly decoupled. It's just a matter of time, patience and, maybe, a whole carton of 0.1uF capacitors.

Again, let me say that you must not hedge on this problem. If you are getting frustrated now, imagine what it would be like to spend a whole evening entering a long program, only to have it blow up in a couple of microseconds because you ignored the problem at this point. Now that's *real* frustration!

After you get the TEST program running, load the TESTO program and run it. If all goes well, you are all set. It's time to enjoy the fruit of all your labor. Try this program:

```
                BLKT:
000   21 00 08        LXI   H,PORTO    ;SET PORT POINTER TO
                                        ;PORT 0
003   AF        ;STA  XRA   A          ;CLEAR A TO 0'S
004   77        ;MOA  MOV   M,A
005   06 FF           MVI   B,FFH      ;INITIALIZE B
007   0E FF     ;INC  MVI   C,FFH
009   16 FF     ;IND  MVI   D,FFH


00B   15        ;DCD  DCR   D
00C   C2 0B 00         JNZ   DCD        ;IF NOT ZERO,
                                             JUMP TO DCD
00F   0D                DCR   C          ;ELSE DECREMENT C

010   C2 09 00          JNZ   IND        ;IF NOT ZERO,
                                             JUMP BACK TO IND
013   05                DCR   B          ;ELSE DECREMENT B
014   C2 07 00          JNZ   INC        ;IF NOT ZERO,
                                             JUMP BACK TO INC
017   FE FF             CPI   FFH        ;ELSE COMPARE WITH 1'S
                                        ;IF ALL 1'S, Z
                                        ;IF 0'S, NZ
```

```
019   CA 03 00      JZ    STA       ;IF ALL 1'S,
                                    JUMP BACK TO STA

01C   3E FF        MVI   A,FF      ;ELSE SET A TO ALL 1'S
01E   C3 04 00     JMP   MOA       ;AND JUMP BACK TO MOA
                                    :END OF BLKT
```

This program causes the motor status lamps and relays to switch on and off alternately. It is a looping program which should run indefinitely, and the only way to stop it is by going to PROGRAM and switching on the RESET switch. Of course, it can be started again by going to RUN and turning off the RESET switch, assuming, of course, there is no interruption in the power supply to mess up the information you have placed into the program RAM.

Let this program run for an extended period of time. You wouldn't be all crazy to let it run for 24 hours or more. Let the system "burn in" with this program. If you are going to have problems with more extensive programs in the future, it would be nice to iron them out now.

INSTALLING RODNEY'S BUMPER AND CHARGE RINGS

If you haven't figured it out by now, you ought to realize that Rodney needs a pair of passive wheels to balance him. I used a set of casters which are available in just about any hardware or department store. The arrangement isn't really critical, as long as the passive wheels hold Rodney level and don't impede his motion.

The job at hand now is to install the charge rings. These two rings of aluminum, one for (-) BATC and another for (+) BATC, will contact a set of identical rings on the nest assembly. Of course, the rings on the nest assembly are permanently wired to the battery charger assembly. Rodney's charge rings run completely around the outside edge of his skirt. The rings are parallel to one another and must be lined up precisely so they mate with their counterparts on the nest.

You will have to visit a good home improvement store to find the special materials needed here. For one thing, you will need some sections of vinyl "mop board" material, the sort of material normally cemented to walls where the walls meet the floor. Some mop boards are made of wood. Of course, that won't do, so look for the very flexible, vinyl type. You will need enough to wrap all the way around Rodney's skirt and an identical skirt assembly on the nest. This vinyl mop board is available in a standard 2-in. width, so it fits the skirt assemblies perfectly.

Next, you will need some vinyl trim about ⅞-in. wide. It doesn't have to be anything fancy; simply the kind of trim that is sometimes used for covering seams in wooden paneling. It usually comes in 8-ft. sections, and you will need enough to go around Rodney twice and the nest twice. And finally, there is a need for some bands of standard 1/16-in. thick aluminum sheet. These should be long enough, or at least there should be enough to add up to the right length, to go around Rodney and the nest two times.

Figure 9-5 is a view of how all these different materials will be assembled. The "mop board" is the backing for the whole affair, and it is to be bolted to the 3-½ in. sections of the aluminum skirt.

After this backing is bolted into place, the two sections of trim are cemented to it. You can use a good epoxy cement, although vinyl tile cement ought to work just about as well.

The aluminum bands are then fastened to the trim pieces and "mop board" backing with flat head machine screws. You'll have to countersink the holes for these screws so that the charge rings have a uniformly smooth surface.The lower charge ring should be screwed onto place where the 2-in. gaps occur on the skirt assembly. This will be the (+) BATC ring, and is "hot" relative to the COMM potential of Rodney's mainframe.

Since the upper charge ring will carry the COMM potential from the battery charger, the screws for mounting this ring can go through all the vinyl sections and the skirt as well.Although this scheme seems to be quite awkward at first, it turns out to be very sturdy when it's completed.

Any one of the screws to the lower charge ring can serve as a connecting point for a length of 18 AWG wire to (+) BATC on the

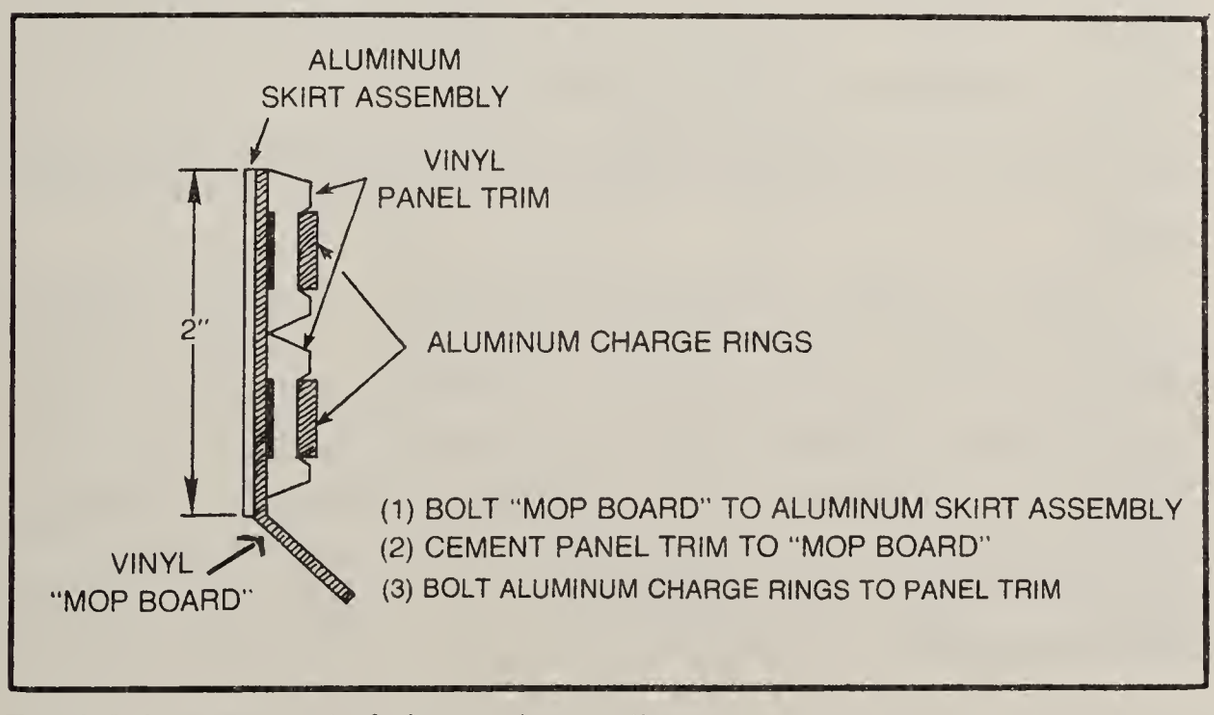


Fig. 9-5 Arrangement of charge-ring sections.

power-distribution panel. Likewise, any of the screws on the upper ring can be a connector for a COMM lead to (–) BATC on the power panel.

THE NEST ASSEMBLY

Now, all you have to do is construct the nest mainframe. You can begin by building a circular bottom plate and skirt assembly that is identical to the one that Rodney already has. For the nest, however, the bottom plate can be made of wood or some other material that is easier to work with than aluminum. Suit yourself.

Fashion the skirt from aluminum as described in Chapter 4, and attach the charge-ring assembly, one practically identical to Rodney's. The only difference is that the lip extending from the vinyl mop board should be at the top, rather than at the bottom as shown in Fig. 9-5.

Now, it's time to make some legs for the nest. Half-inch threaded stock works nicely here because it lets you assemble legs with adjustable lengths. It is important to be able to adjust the height of the charge rings on the nest so that they mate perfectly with Rodney's charge rings.

Simply cut the threaded stock into four lengths approximately 6-in. long, drill four holes at "corners" of the nest's bottom plate, and feed the threaded stock through these holes. Bolt the "legs" into place with a pair of nuts on each one, one on top of the plate and one on the bottom.

You might like to fit some vinyl tips on the ends of the legs that make contact with the floor. It isn't such a good idea to have the legs on the nest scratching the floor or getting snagged on the carpet. Look around a department or hardware store for some soft "feet." I used the little tips that come with spring-loaded curtain rods but since that meant throwing away a couple of potentially useful curtain rods, you might want to use something else.

At any rate, adjust the charge-ring and nest assembly so that the rings line up with Rodney's rings and, of course, are level all around.

Finally, complete the nest assembly by installing a battery charger and current limiter in the nest. I have a 4-ampere charger acquired from Sears. Such chargers should not be allowed to deep-charge Rodney's battery on a routine basis—that will destroy the battery in a short period of time. Couple the output of the battery charger through a current limiter such as Sears' Ampere Damper ™. Using this device, the charging current can be adjusted to suit the battery you buy.

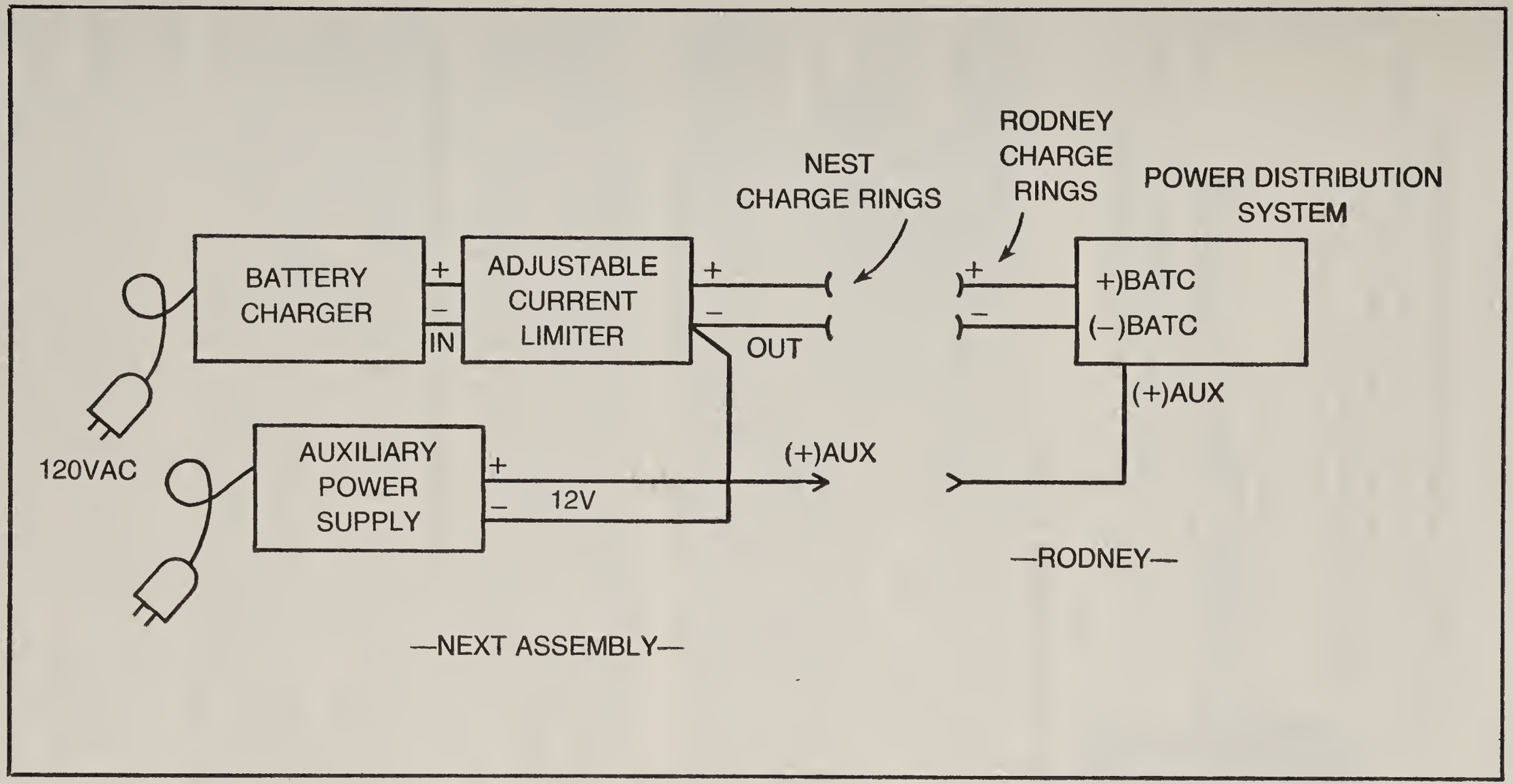


Fig. 9-6. Block diagram of the battery-charger system.

Table 9-2. Electrical components for the battery-charger system.

1 12V, 14-Ahr motorcycle battery (Sears #28K44314N) 1 12, 4A battery charger (Sears #28K7126) 1 battery-charger current limiter (Sears #28K7104) Equivalent devices from other sources may be used—Sears is merely a suggested source.

Connect the negative output of the current limiter to the upper charge ring on the nest assembly and wire the positive output to the lower charge ring.

THE BATTERY-CHARGER SYSTEM

Figure 9-6 shows the electrical features of the battery system in a schematic format. Naturally there has to be some sort of battery in the scheme, so let's give that some consideration now.

I selected a 12V, 14-AHr motorcycle battery for my own Rodney unit. It fits nicely in the space below the circuit cards and it has an amp-hour rating that allows Rodney to run about 4 hours at a time before needing another charge.

By the time you begin reading about this battery, the newer gel batteries might be available at a reasonable cost. At the time of this writing, their higher cost makes it hard to justify the special convenience.

A battery, whether it is lead-acid or gel, should not be recharged routinely at a rate exceeding 10% of its rated ampere-hour discharge rate. My own battery has a 14-AHr rating, which means I have to keep the current limiter set at 1.4A. If you happen to use a 10-AHr battery, you'll have to set the current limiter down to 1A.

Table 9-2 lists the components used for the entire battery-charger scheme. Even if you have no immediate plans for running Rodney from the battery, it is a good idea to connect the battery system now. Much of the work in upcoming chapters involves storing some very extensive programs in the program RAM section. The slightest interruption in supply power will wipe out all that programming. So the battery, battery charger, and auxiliary power supply should be connected and in full service at all times.

If anything shuts down utility power in your lab, the battery will continue supplying power to the system; if the motors aren't running, the battery will preserve the contents of the program RAM for quite a number of hours. With the auxiliary power supply also connected to the system, it can supply power to the program RAM if anything should happen to go wrong with the battery system. So everything is backed up pretty well with this scheme.

Running Alpha Rodney

We have been fooling around long enough. It's high time we get something rolling around the floor and behaving in a semi-intelligent manner. After completing the work and conducting the experiments outlined in this chapter, you will have been exposed, first hand, to a bit of genuine machine intelligence.

Recall that the essence of an Alpha-Class machine is its purely reflexive and, for the most part, random behavior. Alpha Rodney will behave much as a little one-cell creature that struggles to survive in its drop-of-water world. The machine will blunder around the room, working its way out of menacing tight spots, and hoping to stumble, quite accidentally, into the battery charger.

This chapter also includes a few circuit refinements that add to Rodney's range of behavior. However, bear in mind the real point of this whole project—to demonstrate the principles of machine intelligence. Dwelling too much on the showy side of the machine will mask the truly remarkable features of its existence.

ALPHA RODNEY-ONE—THE SIMPLEST SYSTEM

The flowchart in Fig. 10-1 summarizes the behavior of the simplest Alpha Rodney system. For the sake of convenient reference, call it *Alpha Rodney-One.*

Study this flowchart carefully before looking at the programs for it. It is important to understand the basic mode of behavior and the definitions of certain special terms before anything else can make much sense.

Definitions of Special Terms and Acronyms

Compare the expressions in Fig. 10-1 with the following list of definitions and preliminary explanations.

INITIALIZE—A general housekeeping operation that initializes the position of the microprocessor stack pointer and sets up

the system for dealing mainly with the Port 0 I/Os—ENVL and ACTL.

FETCH ENVL—A read operation where by the 8-bit, lower-order portion of the environment status word is fetched into the system. After all, Rodney must know what is going on around him before he can take any action to alter his environment.

Note that the FETCH ENVL operation implies that we are working only with the 8 lower-order bits. The entire Alpha Rodney-One scheme, as it is presented here, totally ignores any environmental parameters assigned to the 8 higher-order bits. Only ENVL and ACTL will be used as far as communication with the outside world is concerned. The higher-order portion, ENVH and ACTH at Port 1 will be applied in a later version.

RUN TO LM—The question here is whether or not the left motor is getting a command that tells it to run. The answer is "yes" (Y) if it is seeing a run command, and it is "no" (N) if it is seeing a stop command.

LM RUN—Is the left motor indeed running? This question is answered by testing the motor-control, speed-sensing circuit for the left motor. If the motor is running as it should be, the answer is "yes" (Y). But, if the motor is stalled or else running too slowly for some reason, the answer is "no."

The RUN TO LM and LM RUN questions are closely related to one another, but they serve entirely different purposes. RUN TO LM merely answers this question: Is the motor *supposed* to be running? LM RUN asks: Is the motor *actually* running? A discrepancy between the answers to these two questions signals a problem as far as Alpha Rodney is concerned.

RUN TO RM—This statement has the same general meaning as RUN TO LM. RUN TO RM, of course, applies to the right motor, rather than the left one.

RM RUN—Again, the expression serves the same function as its left-motor counterpart, LM RUN.

FEED—Is the system feeding at the battery charger? If Rodney is connected to the battery charger and is getting energy from it, the answer to this question is Y. Otherwise, the result is N.

Recall that the FEED signal is included in the ENVL status word. It originates at the charge rings on the robot and takes on an active-high, logic-1 level only when battery-charger power is present. Rodney, you see, can be connected up to the battery charger, but if the charger is turned off (or the connection is a poor one), FEED drops to logic 0 and the answer to the question is "no" (N).

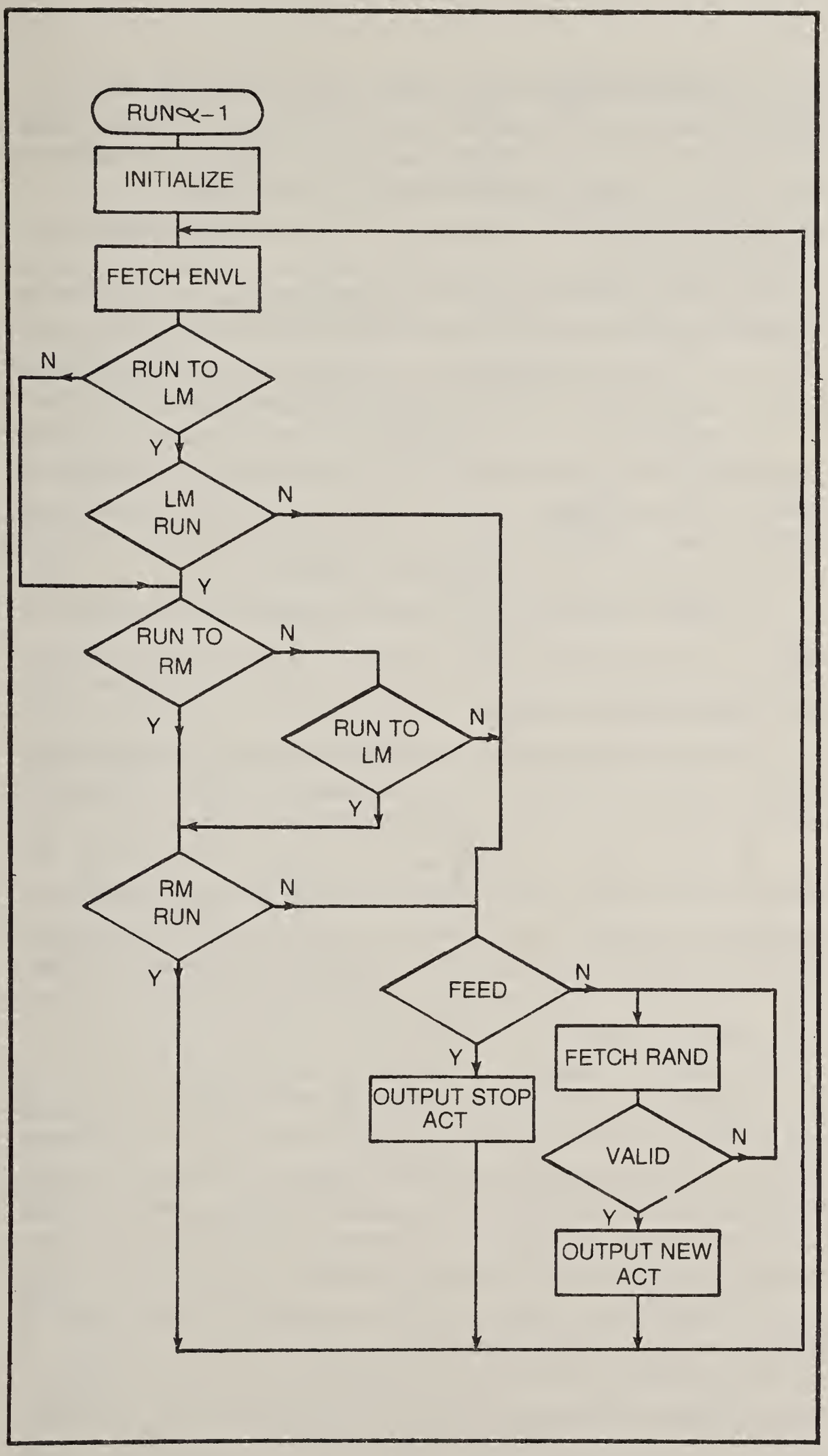


Fig. 10-1. Alpha Rodney-1 flowchart.

OUTPUT STOP ACT—This operation sends a motor-stopping code to ACTL. The purpose is to turn off both motors. Although the drive and steering code format includes four different stop codes, B will be used because of its simplicity.

FETCH RAND—This means, "Fetch a 4-bit motor-control code from Port 2."

VALID—While any number fetched from the random number generator at Port 2 could be successfully used by the motors, we must impose some restrictions as an "intuitive" element of Rodney's behavior. In this particular scheme, standing still cannot effectively solve a stall problem. So if the FETCH RAND operation should happen to generate one of the four stop codes, the code must be replaced with a different one, one that causes Rodney to move in some fashion. VALID merely asks the question: Is this a motor-control code that will result in some motion?

OUTPUT NEW ACT—This command sends a new motor-control code to Port 0, ACTL.

The Main Flowchart Analysis

The Alpha-One system is first initialized to set the stack pointer and inform the system that most I/O operations will be taking place at Port 0. This is a one-time INITIALIZE operation.

The first really meaningful step is to query the relevant elements of the environment. FETCH ENVL acquires information regarding the control codes that have been previously directed to the motors (bits D0 through D3), stall status for both motors (D4, D5), and FEED (D6). The remaining bit (D7) in ENVL is not relevant in this case.

Information contained in ENVL is then the foundation for answering a series of questions. First, RUN TO LM: Do the control bits to the left motor tell it to run? If so, the system determines whether or not the motor is actually running (LM RUN). If the left motor is not receiving a code that calls for a running operation *for that motor*, the system then looks to see if the right motor is supposed to be running (RUN TO RM). If it turns out that the right motor is also getting a code that calls for a running operation (Y from RUN TO RM), then that motor is tested to see if it is actually running.

Suppose the motor system is receiving valid "run" commands and both motors are indeed running. The answer to all four questions is "yes," and the whole business cycles back to check ENVL once again. No matter what the combination of running codes might be, the system shows that the machine will continue doing the same thing, at least until one of the four questions is answered with a "no."

Now, assume the left motor is supposed to be stopped, and the right motor is supposed to be running. This situation arises whenever Rodney is to turn to the right.

The answer to RUN TO LM, in this instance, is N. If the left motor isn't supposed to be running, there is little point in actually checking it. A more sophisticated program might call for checking the actual motor status against a N from RUN TO LM, but all that would accomplish is sensing whether the motor is being pushed by some outside force or, perhaps, rolling down a hill.

So if the left motor is not supposed to be running, the next step is to see if the right motor is supposed to be running. In the present example, the answer from RUN TO RM should be Y. Then the system checks the actual operation of the right motor at RM RUN. Again, if the answer is Y, the whole operation starts over again by picking up an updated version of ENVL.

The system works much the same way if the right motor is supposed to be stopped and the left motor is supposed to be running. This would be the case for left-hand turns. The reply to the first RUN TO LM is Y, the left motor is supposed to be running. The motor is then tested by LM RUN to see if it is indeed running, and if the answer is Y, the system looks at the condition the right motor is supposed to have. In this particular example, the answer should be N.

The three examples cited thus far assume everything is in good operating order, at least one of the two motors is running and nothing very dramatic happens. The system simply keeps looping around to FETCH ENVL time and time again.

Now, several things can go wrong to upset this tidy loop, and no matter what the nature of the problem might be, the final result is a look at the FEED system. Suppose the left motor is supposed to be running, but it isn't, as signalled by a N output from LM RUN. This situation breaks up the basic loop and sends the operation to FEED. The same sort of thing happens if the left motor is supposed to be running, but RM RUN indicates it is not. That also calls up the FEED step. And finally, it is possible to break up the basic loop and go to the FEED question. This happens if both motors are being told to stop. In this case, the N output of the first RUN TO LM takes the operation directly down to RUN TO RM. If the left motor is not getting a valid run code, the next step is to see if the right motor is getting a valid run code. The N from RUN TO RM leads to another inquiry regarding the control code to the left motor; if the motor isn't supposed to be running, the system goes to the FEED question.

The second look at RUN TO LM is necessary for the simple reason that RUN TO RM cannot tell whether the left motor is supposed to be running or not. The Y and N replies from the first RUN TO LM are joined together (thus indistinguishable) by the time they reach RUN TO RM.

In summary, the system runs in a loop as long as at least one motor is getting a valid run code and the motor(s) that is(are) supposed to be running is(are) doing precisely that. The system immediately checks FEED, however, if either motor is stalled or a set of motor-control commands calls for stopping both motors at the same time.

FEED is a question that is easily answered by the information contained in ENVL—it is the FEED bit from the battery-charger terminals. If the system answers FEED with a Y, a stop code is loaded to ACTL. If the machine happens to be trying to move at the time, it is told to relax and stop trying.

If, on the other hand, the reply from FEED is N, the system goes to FETCH RAND. This picks up a random 4-bit number from Port 2 and then checks it at VALID to make certain the random number does not represent a stop code. Should the new code be a stop code, the N line from VALID leads back to FETCH RAND where the system is told to pick another number. The system continues picking new numbers until a number, other than one representing a stop code, occurs and the answer to VALID is Y.

OUTPUT NEW ACT then loads the 4-bit number as the motor-control portion of ACTL, and the machine is told to do some sort of motion. Whether or not this new, randomly selected motor code actually clears the problem is determined by cycling back to FETCH ENVL and starting all over again. If the new code works, the system enters its most desirable loop. If the new code doesn't work, it doesn't take long for the operations to return to FEED where there is a chance a new random number will be selected and tried.

We could now delve into some of the finer features of the flowchart in Fig. 10-1, but it might be more fun and equally instructive to consider some commonly asked questions about the scheme.

What will Alpha Rodney-One do when the system is first started? A certain machine code could be included in INITIALIZE. We could, for example, output a fast-forward command to ACTL as part of INITIALIZE. But, don't do it! This sort of thing borders on heresy. An intelligent machine must be told to do something specific as rarely as possible. It is bad enough that the scheme already contains such a step in the form of OUTPUT STOP ACT.

The machine isn't "told" what to do initially. The registers are going to come up with garbage in them, and Rodney is either going to move in one way or another or, if the "garbage" happens to be a stop code, he will stand still only for the few milliseconds it takes the system to correct this particular state of affairs.

So what does Rodney do when he is first turned on? For all practical purposes, he takes off across the floor. Just be sure to keep your own feet out of the way when you throw the RESET switch off.

What happens when Rodney makes contact with his battery charger? The event is first seen as a stall condition. One or both motors will stall whenever Rodney first encounters the nest assembly. What happens after that depends on whether or not the electrical connection to the charger is a good one.

The FEED question in the flowchart in Fig. 10-1 is most relevant to this particular situation. Anytime Rodney stalls, his first question is, "Am I stalled at my own battery charger?" If the stall isn't due to the nest, or if the connection to the battery charger is a poor one, FEED outputs a "No" answer. If, on the other hand, the stall is at the nest and the connection to the battery charger is a good one, the answer from FEED is "yes" and Rodney responds by switching off his motors.

How long does Rodney stay at the battery charger? He stays there until the connection is somehow interrupted. Left to his own devices, Rodney will remain there forever. If the connection is interrupted somehow, the output of FEED is no longer Y, and the system responds immediately by picking up a response from FETCH RAND. In fact, one way to keep Rodney quiet and out of the way is to lead him up to the battery charger. He will remain there, quietly absorbing energy for the battery as long as the supply is available. Turn it off, even for just an instant, and he's off and running.

My own Alpha Rodney-One spends most of his time resting at the charger; when I need him for a few hours every day or so, the battery is fully charged. The battery cannot be "overcharged" if you are using the current-limiting circuit described in an earlier chapter.

What keeps Rodney from running down his battery before finding the nest and battery charger? Nothing. There is no way this Alpha-One system can sense the presence of the charger until Rodney makes direct contact with it.

Rodney is an exceedingly active creature, however, and if his movement is restricted to an average-size room, and the nest is centrally located, he will most likely find it before it's too late. My own unit has never failed to stumble across the charger in time. In

fact, he usually "finds" it too soon. If your Rodney is spoiling the fun by finding the nest too often, simply switch off the power to the battery charger. The nest will then seem to be just another obstacle to be dealt with.

The XM RUN Routine

Most of the programming for the steps in Fig. 10-1 is quite simple and straightforward. A notable exception, however, is the two run-sensing steps, LM RUN and RM RUN. These two operations are practically identical, but it turns out that they are more complex than the main Alpha-One routine.

Recall that the purpose of the LM and RM RUN routines is to sense whether or not the respective motors are indeed running when they are supposed to be running. The process begins by sensing whether or not a light passing through some holes in a motor gear is flashing. If the light is flashing, the gear (and motor) is turning, but if the light is shining steadily or if there is no light at all, the motor certainly is not moving.

Rather than building and installing a special circuit for detecting the flashing (or non-flashing) light from the motor speed-sensing circuit, we are going to use a software subroutine. The only hardware that is dedicated to this particular application is an LED and phototransistor located on or near the gearbox for each motor.

The software operation includes:

- turning on the light to the motor that is to be checked
- picking up a bit from ENVL that signals whether or not the phototransistor is seeing a flashing light
- counting the number of on/off transitions that occur in a given period of time and
- generating a flag bit that signals whether or not the motor is turning at a speed fast enough to be considered "OK."

A rather detailed flowchart for the LM and RM RUN operations appears in Fig. 10-2. For all intents and purposes, the operations for the two units are identical, so a look at the theory of operation for one of them is sufficient. The differences that do exist will be reflected in the actual programming.

This particular flowchart represents a program subroutine called XMRN, where the "X" stands for either an "L" for left motor sensing or an "R" for right motor sensing. So don't let this bit of nomenclature throw you. Some of the other terms on this flowchart require some special explanation, though.

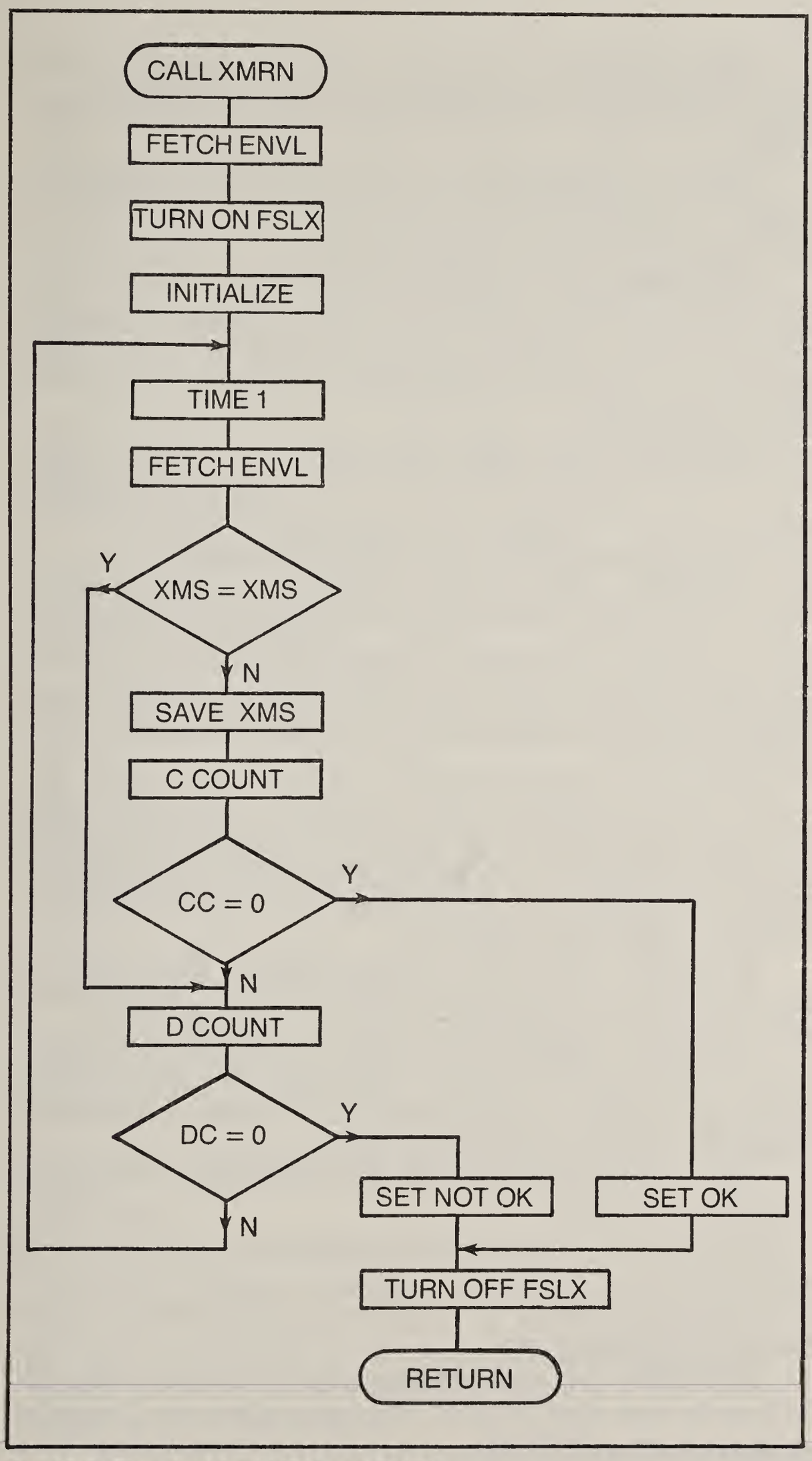


Fig. 10-2. Flowchart for LMRN and RMRN.

TURN ON FSLX—This means, "Turn on the lamp for either the left or right motor." The lamps are turned on only when they are needed.

TIME 1—A software time delay of about 0.5 seconds. Its purpose is to give the motor a chance to turn a bit before checking the output of the phototransistor another time.

XMS=XMS—The question is whether or not the status bit from the phototransistor is the same as it was on the previous query. If there has been no change in the light level, the answer is Y. If the light level has changed, presumably because the motor turned a little bit, the answer is N.

XMS SAVE—The XMS=XMS operation just described calls for a bit that was gathered and stored on an earlier pass through the operating cycle. The XMS SAVE operation is responsible for storing the phototransistor status bit from the present cycle.

C COUNT—The criteria for whether or not the motor is turning fast enough to be considered running properly is based on a counting scheme. The scheme includes two counters, C COUNT being one of them. In this case, the counter keeps track of the number of times the system senses a change in light level from one cycle to the next.

CC=0—C COUNT is actually a down counter, so each time there is a change in light level, the count decrements by one. When the count reaches zero, the system knows a significant number of pulses has occurred. A Y output indicates CC has reached zero, while an N output says it has not.

D COUNT—This is the second counter in this scheme. D COUNT keeps track of the total number of samples, whether they show any change in light level from the phototransistor or not.

DC=0—D COUNT is also a down counter. If its count ever reaches zero, there is a good chance the motor is not running, or at least it isn't running as fast as it should.

SET NOT OK—Set the flag that indicates the motor is not running as it should be.

SET OK—Set the flag that indicates the motor is running properly.

TURN OFF FSLX—Turn off the lamp that was used for the motor speed-sensing tests.

RETURN—This entire flowchart represents a subroutine called from the main Alpha program. Since it is a subroutine, it must be terminated with a RETURN.

Those are the special terms used on the flowchart in Fig. 10-2. FETCH ENVL and INITIALIZE have the same meaning they do on the main program flowchart in Fig. 10-1.

Suppose it is time to call this subroutine. Again, it might be for either left or right motor tests, it doesn't make any difference at this point in the explanation. The first step is to sample the low-order environment with FETCH ENVL, with the appropriate test lamp turned on by TURN ON FSLX. TURN ON FSLX is actually an ACTL loading operation involving only the FSLL or FSLR bit (D4 or D5). The bit representing the lamp to be tested is changed to a logic-0 level, while the other bits are allowed to remain as they were from the FETCH ENVL.

The two counters, C COUNT and D COUNT, are initialized at the INITIALIZE operation. A 4-bit register, actually the four higher bits of C COUNT, is also cleared at this step. This little register holds the XMS flags.

Immediately after initializing the system, TIME 1 inserts a short delay, about 0.5 seconds. After the delay the system checks the environment again. This time the purpose is to see what the phototransistor is doing.

The phototransistor status bit (XMS) is normally compared with that of the previous cycle, but since the first cycle has not yet been completed at this point in the explanation, it is compared with the 0 value set during the initialization step. If this XMS=XMS shows no change, the old and new XMS bits are the same, the indication is that the motor has not turned. (Again, this is not a valid conclusion on the first cycle, but any error that might occur now will be wiped out by an overwhelming amount of good data later on.) So if there is no change, the Y output of XMS=XMS skips down to D COUNT. The reason for this will be explained in a moment.

In the meantime, assume the old and new XMS values are not equal. This indicates (on all but this first pass) that the motor has turned and the N output leads to a C COUNT operation. Each time the system picks up an XMS that is different, the new value is saved somewhere among the four higher-order bits in C COUNT. C COUNT is then decremented one count, indicating a change has taken place.

C COUNT is normally initialized to something on the order of 10. If four transitions are noted by the system, C COUNT reaches zero; CC=0 senses this condition. On our first pass through the flowchart, however, the count cannot possibly count down to zero. In fact, the best it can do is decrement from 4 to 3 which means the output of CC=0 has to be N.

D COUNT is decremented each time an XMS=XMS operation occurs, no matter what the output might be. It might decrement as the result of a Y output from XMS=XMS or, more indirectly, as the

result of an N output and some C COUNT operations. The essential point is that D COUNT keeps track of the number of times the whole subroutine is cycled.

D COUNT must be initialized at the beginning with some number that is larger than the C COUNT number. Loading D COUNT with 8_{10} counts would be compatible with 4 counts initially loaded into C COUNT.

We are getting very close to showing exactly how the counters and, in fact, the entire scheme works. Suppose that DC=0 results in an N output—the D COUNT has not counted down to zero. If this is the case, the entire testing sequence begins once again from TIME 1. As the system cycles around time and again, C COUNT decrements each time it picks up a change in the XMS bit, but D COUNT decrements every time around.

Here's the critical question: Which one of the two counters will reach zero first? Remember that D COUNT is initialized with a number that is larger than the one loaded into C COUNT. Well, the answer depends on how fast the motor is running. If the motor is not running at all, C COUNT never decrements and D COUNT will certainly reach zero first. In that case, the NOT OK flag is set, the LED is switched off, and the program returns to the main Alpha-One routine, carrying the NOT OK flag with it.

If, on the other hand, the motor is running properly, C COUNT will decrement to zero before D COUNT does. As a result of a Y from CC=0, the OK flag is set at SET OK, the LED is turned off, and the subroutine returns to the main program with an OK flag. Whenever this subroutine is called again, everything is initialized and the counters race each other to zero.

Before going into the programming details, consider the effect TIME 1 has on the overall system. This short delay occurs each time XM RUN executes one of its internal loops. The number of looping operations depends on the number initially loaded into the counter that reaches zero first. If, for instance, C COUNT is initialized at 4 and the motor is running properly, the execution time of the subroutine is 4 times the value of the time delay from TIME 1. A 0.5-second delay at TIME 1 would be a total execution time of about 2 seconds. If a motor is running, it takes about 2 seconds to determine this fact. If the motor is not running, D COUNT has to decrement about 8 counts before the cycling can be completed, and that takes about 4 seconds.

Time Delays In Perspective

At first thought, these rather long execution times for the XM RUN subroutine might seem to pose something of a problem, espe-

cially when considering that each moment of stall time drains an excessive amount of valuable power from the battery. Of course, the TIME 1 interval can be shortened. The initial numbers loaded into C COUNT and D COUNT could then be reduced. Before tampering with the specifications listed here, however, look at the overall picture.

First, consider that these time delays aren't at all relevant while Rodney is running freely around on the floor. The delays do indeed occur, but since Port 0 has an active motor-control code stored in it, the delays are never noticed.

The delays are noticeable whenever a stall condition occurs. The delay is no longer than one D COUNT cycle, though, because the flowchart in Fig. 10-1 shows that only one of the two XM RUN subroutines is executed during a stall. The delay in this case is 4 seconds at most.

TIME 1 delays show up in a somewhat different fashion when Rodney is trying to clear up a stall condition and the code he is trying isn't working. Under this set of circumstances, the D COUNT and TIME 1 features give the new code up to 4 seconds to work. If it doesn't work within this amount of time, it's no good anyway and Rodney picks another motion code.

For experimental purposes, the TIME 1 delay interval can be shortened, but the initial values of D COUNT and C COUNT cannot be reduced significantly without reducing the reliability of the motion-sensing scheme.

Table 10-1. Summary of Alpha Rodney-1 I/O and subroutine addresses.

Hardware-Oriented Addresses
PORT-0 (ENVL/ACTL) 0800H
PORT-2 TSWR 0802H

Right Motor Motion Codes:	RMF—D3 of Port-0
	RMR—D2 of Port-0
Left Motor Motion Codes:	LMF—D1 of Port-0
	LMR—D0 of Port-0
Motor Stall Codes:	LMS—D4 of Port-0 ENVL
	RMS—D5 of Port-0 ENVL
Stall Test Lamps	FSLL—D4 of Port-0 ACTL
	FSLR—D5 of Port-0 ACTL

Software-Oriented Addresses
TOPS (Top of Stack)—03FFH
ALPH1 (Main Alpha Rodney-1 program)—0000H
RTLM—0100 (RUN TO LM)
RTRM—0110 (RUN TO RM)
LMRN—0150 (LM RUN)
RMRN—0250 (RM RUN)
TIME1—0300
FETR—0350

THE ALPHA RODNEY-ONE PROGRAM

Unfortunately, a complete commentary on the programming steps is not appropriate for a book of this type. The program is presented in something of an assembly language format that includes brief step-by-step comments. Experimenters who have acquainted themselves with the 8085 instruction set will be able to determine the exact procedures for themselves and even insert modifications and some improvements.

The programs and subroutines ought to be studied, at least comparing them with the flowcharts, before beginning the actual programming operation. This will certainly give you a better appreciation for what is happening and, more importantly, provide the insight you will need for any software troubleshooting that might be necessary.

```
                        ALPH1:
                        ;BEGIN INITIALIZE
000  31   FF 03         LXI SP,   TOPS     ;INITIALIZE STACK
                                           ;AT TOP
003  21   00 08         LXI H,    PORTO    ;INITIALIZE H,L FOR
                                           ;PORT-0 OPERATIONS
006  7E          ;FETEN MOV       A, M
007  CD   00 01         CALL      RTLM     ;CALL "RUN TO LM"
00A  CA   16 00         JZ        RURM     ;IF N, JUMP TO "RUN
                                           ;TO RM"
00D  CD   50 01         CALL      LMRN     ;ELSE CALL "LM RUN"
010  79                 MOV       A,C      ;FETCH STALL FLAG
011  E6   10            ANI       10H      ;ISOLATE STALL FLAG
013  CA   29 00         JZ        FEED     ;IF N, JUMP TO "FEED"
016  CD   10 01 ;RURM   CALL      RTRM
019  C2   3A 00         JNZ       RULM2    ;IF N, JUMP TO SECOND
                                           ;"RUN TO LM"
01C  CD   50 02 ;RMN    CALL      RMRN     ;ELSE CALL "RM RUN"
01F  79                 MOV       A,C      ;FETCH STALL FLAG
020  E6   10            ANI       10H      ;ISOLATE STALL FLAG
022  CA   29 00         JZ        FEED     ;IF N, JUMP TO "FEED"
025  C3   06 00         JMP       FETEN    ;ELSE RETURN TO
                                           ;"FETCH ENVL"
028  00                 NOP
029  7E          ;FEED  MOV       A,M      ;FETCH ENVL
02A  E6   40            ANI       40H      ;ISOLATE FEED BIT
02C  CA   34 00         JZ        FTR      ;IF N, JUMP TO "FETCH
                                           ;RAND"
02F  36   F0            MVI       M,F0H    ;"OUTPUT STOP ACT"
031  C3   06 00         JMP       FETEN    ;RETURN TO "FETCH ENVL"
034  CD   50 03 ;FTR    CALL      FETR
037  C3   06 00         JMP       FETEN    ;RETURN TO "FETCH ENVL"
03A  CD   00 01 ;RULM2  CALL      RTLM
03D  C2   1C 00         JNZ       RMN      ;IF Y, JUMP TO "RM RUN"
040  C3   29 00         JMP       FEED     ;ELSE JUMP TO "FEED"
                                           ;END
```

This completes the main Alpha Rodney-One program, but it cannot be properly run until the called subroutines are programmed into the system. The subroutines required are as follows: RTLM (RUN TO LM), RTRM (RUN TO RM), LMRN (LM RUN), RMRN (RM RUN), and FETR (FETCH RAND).

Subroutines RTLM and RTRM

```
                                        RTLM:
                                                    ;START WITH STATUS WORD
                                                    ;IN REGISTER A
100  47                          MOV    B,A         ;SAVE A IN B
101  87                          ADD    A           ;SHIFT LEFT TO LINE UP
                                                    ;LMR AND LMF
102  A8                          XRA    B           ;EXOR TO COMPARE BIT
                                                    ;VALUES
103  E6   02                     ANI    02H         ;ISOLATE LM BIT
                                                    ;IF Y, THEN NZ
                                                    ;IF N, THEN Z
105  C9                          RET                ;RETURN
                                                    ;END OF RTLM
                  RTRM:
                                                    ;START WITH STATUS WORD
                                                    ;IN REGISTER A
110  47                          MOV    B,A         ;SAVE A IN B
111  87                          ADD    A           ;SHIFT LEFT TO LINE UP
                                                    ;RMR AND RMF
112  A8                          XRA    B           ;EXOR TO COMPARE BIT
                                                    ;VALUES
113  E6   08                     ANI    08H         ;ISOLATE RM BIT
                                                    ;IF Y, THEN NZ
                                                    ;IF N, THEN Z
115                              RET                ;RETURN
                                                    ;END OF RTRM
```

Subroutines for LMRN and RMRN

Consider the following register definitions while studying this subroutine: C COUNT is in D0-D3 of register C, XMS status memory is in D4-D7 of register C, D COUNT is in D0-D3 of register D, and OK/NOT OK flag is D4 of register C.

```
                                        LMRN:
150  7E                          MOV    A,M         ;"FETCH ENVL"
151  E6   EF                     ANI    EFH         ;SET FSLL BIT TO ZERO
153  77                          MOV    M,A         ;"TURN ON FSLL"
154  0E   04                     MVI    C,04H       ;INITIALIZE C COUNT
                                                    ;AND LMS STATUS MEMORY
156  16   08                     MVI    D,08H       ;INITIALIZE D COUNT
                                                    ;
158  CD   00 03 ;CTIM            CALL   TIME1
15B  7E                          MOV    A,M         ;'FETCH ENVL"
15C  A9                          XRA    C           ;EX-OR TO COMPARE BIT
                                                        VALUES
```

```
15D  E6  10          ANI   10H     ;ISOLATE LMS BIT—IF
                                   ;SAME,Z. IF NOT SAME,
                                        NZ
15F  CA  6B 01       JZ    DCONT   ;IF Y, THEN JUMP TO
                                   ;"D COUNT"
162  A9              XRA   C       ;ELSE RESTORE NEW
                                        LMS BIT
163  4F              MOV   C,A     ;"SAVE LMS"
164  E6  0F          ANI   0FH     ;ISOLATE C COUNT
166  3D              DCR   A       ;DECREMENT A
167  CA  76 01       JZ    SETOK   ;IF Z, JUMP TO "SET OK"
16A  0D              DCR   C       ;"C COUNT"
16B  15      ;DCONT  DCR   D
16C  C2  58 01       JNZ   CTIM    ;JUMP BACK TO "TIME 1"
                                        IF N
16F  0E  00  ;SNOK   MVI   C,00H
171  7E      ;TOFL   MOV   A,M
172  F6  F0          ORI   F0H     ;SET FSLL BIT TO ONE
174  77              MOV   M,A     ;"TURN OF FSLL"
175  C9              RET           ;RETURN—D4 OF C IS NZ
                                   ;IF LEFT MOTOR IS
                                        TURNING,
                                   ;ELSE Z
176  OE  10  ;SETOK  MVI   C,10H   ;
178  C3  71 01       JMP   TOFL    ;JUMP TO "TURN OFF FSLL"
                                   ;END OF LMRN


             RMRN:
250  7E              MOV   A,M     ;"FETCH ENVL"
251  E6  DF          ANI   DFH     ;SET FSLR BIT TO ZERO
253  77              MOV   M,A     ;"TURN ON FSLR"
254  0E  04          MVI   C,04H   ;INITIALIZE C COUNT AND
                                   ;RMS STATUS MEMORY
256  16  08          MVI   D,08H   ;INITIALIZE D COUNT
258  CD  00 03 ;CTIM2 CALL  TIME1
25B  7E              MOV   A,M     ;"FETCH ENVL"
25C  A9              XRA   C       ;EX-OR TO COMPARE
                                        BIT VALUES
25D  E6  20          ANI   20H     ;ISOLATE RMS BIT—
                                        IF SAME,
                                   ;Z. IF NOT SAME, NZ
25F  CA  6B 02       JZ    DCONT2  ;IF Y, THEN JUMP TO
                                   ;"D COUNT"
262  A9              XRA   C       ;ELSE RESTORE NEW RMS BIT
263  4F              MOV   C,A     ;"SAVE LMS"
264  E6  0F          ANI   OFH     ;ISOLATE C COUNT
266  3D              DCR   A       ;DECREMENT A
267  CA  76 02       JZ    SETOK2  ;IF Z, JUMP TO "SET OK"
26A  0D              DCR   C       ;"C COUNT"
26B  15      ;DCONT2 DCR   D
26C  C2  58 02       JNZ   CTIM2   ;IF N, JUMP BACK TO
                                        "TIME 1"
26F  0E  00  ;SNOK2  MVI   C,00H   ;
271  7E      ;TOFL2  MOV   A,M
272  F6  F0          ORI   F0H     ;SET FSLR BIT TO 1
274  77              MOV   M,A     ;"TURN OFF FSLR"
```

```
275  C9                      RET             ;RETURN—D4 OF C IS NZ
                                             ;IF RIGHT MOTOR IS
                                                 TURNING,
                                             ;ELSE Z
276  OE  10      ;SETOK2    MVI     C,10H
278  C3  71 02               JMP     TOFL2   ;JUMP TO "TURN OFF FSLR"
                                             ;END OF RMRN
```

TIME I Subroutine

```
                 TIME1:
300  06  FF                 MVI     B, FFH  ;INITIALIZE HIGH-ORDER
                                             ;COUNT REGISTER
302  1E  FF      ;SETE      MVI     E, FFH
304  1D          ;DCRE      DCR     E       ;DECREMENT LOW-ORDER
                                             ;COUNT REGISTER
305  C2  04 03              JNZ     DCRE    ;IF NZ, DECREMENT E AGAIN
308  05                     DCR     B       ;ELSE DECREMENT B
309  C2  02 03              JNZ     SETE    ;IF NZ, JUMP TO
                                                 LOW-ORDER
                                             ;COUNT CYCLE
30C  C9                     RET             ;ELSE RETURN
                                             ;END OF TIME1
```

FETR Subroutine

The FETR subroutine includes selecting a valid random number and outputting it to ACTL as a new response to a stall condition.

```
                 FETR:
350  3A  02 08              LDA     TSWR    ;FETCH RANDOM NUMBER
                                             FROM
                                             ;PORT 2
                                             ;BEGIN "VALID"
353  4F                     MOV     C,A     ;SAVE TSWR IN C
354  CD  00 01              CALL    RTLM
357  C2  64 03              JNZ     ONAC    ;IF Y, THEN JUMP TO
                                             ;"OUTPUT NEW ACT"
35A  79                     MOV     A,C     ;ELSE FETCH TSWR
                                                 FROM C
35B  CD  10 01              CALL    RTRM    ;CHECK RIGHT MOTOR
                                                 CODE
35E  C2  64 03              JNZ     ONAC    ;IF Y, THEN JUMP TO
                                             ;"OUTPUT NEW ACT"
361  C3  50 03              JMP     FETR    ;ELSE FETCH NEW RANDOM
                                             ;NUMBER
364  7E          ;ONAC      MOV     A,M
365  F6  F0                 ORI     FOH     ;SET ACTL
367  77                     MOV     M,A     ;"OUTPUT NEW ACT"
368  C9                     RET             ;RETURN
```

FIRING UP ALPHA RODNEY-ONE

Load the program and subroutines listed in the previous sections. If you have the patience to do so, it is a good idea to run through the addresses to doublecheck the data content of the pro-

gram memory before running the program. Make sure the RESET switch is in its RESET position, throw the RUN/PROGRAM switch to RUN, and *then* turn off the RESET switch.

If the program is loaded properly, Rodney will respond immediately, jumping off into some kind of motion. Let him run for about 10 seconds. If he doesn't run into an immovable object by then, grab him and hold on tightly. In either case, he should start picking random motions to get away from the situation.

If you lift Rodney up from the floor while he is running freely, his motors should continue running in the same directions. Observe Rodney's behavior carefully, making sure both motors are stopping, running forward and in reverse at one time or another. Both motors should stop at the same time only when Rodney is feeding at the nest.

After making certain the nest is energized, coax Rodney toward it. This can be done by grabbing him tightly until he picks a motion code that carries him in the general direction of the nest. Remember that Alpha Rodney-One has no provisions for actively seeking out the nest himself.

Upon making firm contact with the charge rings at the nest, both motors should stop. One way to get him moving again is to pull him away from the nest. Will he immediately return to the nest? That's hard to say; probably not.

A second, and perhaps more meaningful way to get Rodney away from the nest is to turn off the battery charger. He will immediately perceive the nest as an obstacle and start generating random motions to get away from it.

If, heaven forbid, something should go wrong and Rodney's program "spaces out," simply switch on the RESET switch. That will restart the entire program when you subsequently turn off the RESET switch. The RESET switch is Rodney's "panic button." If he starts shaking, buzzing, and clicking and doing all sorts of meaningless things, it is time to hit that panic button.

If the crazy behavior continues, even after trying a couple of resets, it is time to doublecheck the program memory. Put the system into the PROGRAM mode and run through all the addresses relevant to the Alpha-One programming. Correct any errors that might have cropped up in the data portion of the program.

Adding Main Memory And A Look At Beta Rodney

While Alpha Rodney does exhibit some interesting behavioral characteristics, one really has to stretch the definition of *intelligence* to make it fit an Alpha-Class machine. The intelligence is there, of course, but it operates on such a primitive level that little of significance comes from it.

A Beta-Class machine uses the Alpha-Class mechanisms, but extends them to include some memory—memory of responses that worked successfully in the past. The main purpose of this chapter is to extend the intelligence level of your own Rodney machine and get it operating on a Beta-Class level. You are going to be adding more parts to the CPU board, and there is a lot more programming to do but it's certainly worth the effort.

THE MAIN-MEMORY SYSTEM

The main-memory system is something quite different from the program memory you have been using. The program memory is the storage place for Rodney's basic operating programs—programs that are somewhat analogous to intuition or the subconscious in higher-level animals. The main memory is the seat of Rodney's knowledge and, in the case of Beta-Class machines, this means knowledge that is gained only by direct experience with the environment. A Beta-Class machine still relies on Alpha-like random responses in the early going but after experiencing some life and problem solving, knowledge in the main memory becomes dominant over the more primitive Alpha-Class reflex actions.

Fig. 11-1 shows, in a block-diagram fashion, how the main memory fits into the data-bus system already built into the system. The main memory, itself, is composed of eight 1 X 4096 static RAMs, type 2141, to be exact. One byte (8 bits) of data is written into or read out of this main memory system as Port 6 operations. The function enabling signal is $\overline{\text{CSMM}}$ (see Fig. 6-4). Note in Fig.

11-1 that the main memory chips are addressed from a special address bus. This main memory address bus is a full 16-bit bus capable of addressing more than 65k memory locations. Only 4k is specified here, however. The Rodney system can be expanded to accommodate a 65k main memory, and experimenters who have the will and know-how to carry out this expansion are certainly encouraged to do so. However, back to the main memory address bus. A separate bus is needed because it uses too many bits (even at the basic 4k level) to be addressed directly from the microprocessor. Yes, the microprocessor has enough address lines for a 65k main memory, but then there wouldn't be any address space left for the program memory and function-select operations. So we must use a separate address bus.

Now, here's the next question: How can we generate a 16-bit main memory address from an 8-bit microprocessor system? Well, the job requires two data output ports. Look at the 8-bit data latches in Fig. 11-1 labeled as Z120 and Z118. These are actually data output ports 4 and 5 respectively.

Addressing the main memory is thus a 2-step operation: First load the lower 8 bits from the data bus through output port 4 (Z120) and then load the upper 8 bits from the data bus through output port 5 (Z118). A 16-bit address appears on the main memory address bus (MAO-MA15) after those two operations are completed.

As mentioned in the opening chapters of this book, higher order machine intelligence ought to have some sort of short-term memory—a small memory that is capable of remembering what had been happening in the world just before the world undergoes a change. A response that is appropriate at one given moment is often contingent upon what was happening a moment before. The output functions of Ports 4 and 5 serve as the system's short-term memory as well as sources of main-memory address information.

Merely storing this main-memory addressing information is not enough as far as short-term memory functions are concerned. There has to be some way to recover the information without disturbing the main memory itself. This is accomplished by the input functions of Ports 4 and 5. Whenever the system calls for reading the 8 lower bits of main memory address $\overline{\text{IOR4}}$ from the I/O select circuitry goes to logic 0 and main-memory address bits MAO through MA7 go onto the system's data bus. See the block designated Z121 and ½-Z122 in Fig. 11-1.

The 8 higher-order main-memory address bits, MA8 through MA15, are placed onto the data bus whenever $\overline{\text{IOR5}}$ goes to logic 0.

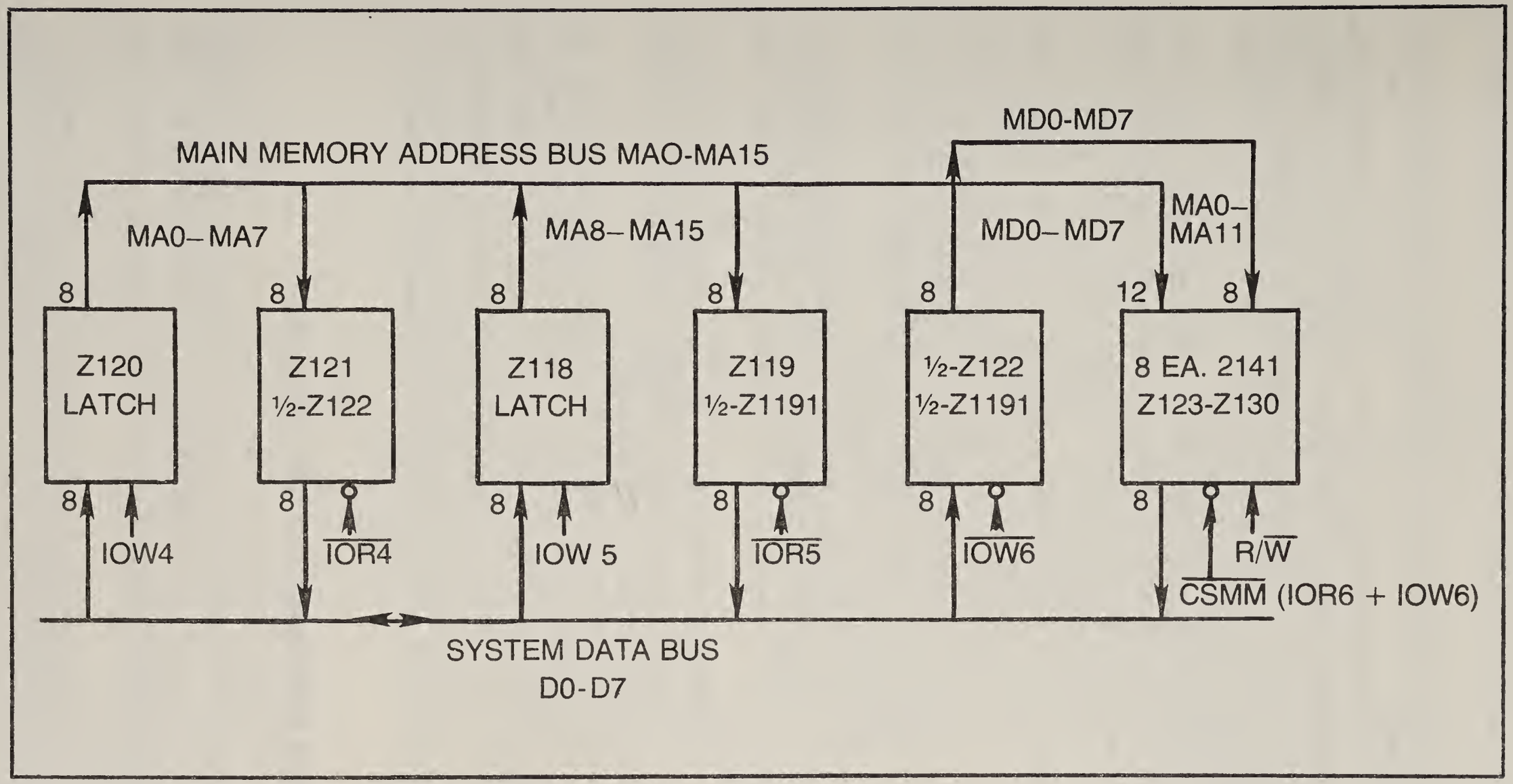


Fig. 11-1. Block diagram of the main-memory hardware.

This Port-5 input function is carried out in the block labeled Z119 and ½-Z1191.

Summarizing the function of Ports 4 and 5, it can be said their output functions both address the main memory and deposit information for short-term memory operations. The only purpose of the input functions of Ports 4 and 5 is to recover short-term memory information whenever it is needed.

In any event, Port 4 handles the lower-order 8 bits, while Port 5 deals exclusively with the higher-order 8 bits. The main memory addressing and short-term memory are 2-byte, 16-bit operations.

Port 6 is divided between a couple of different functions. Its output or *write* function is to provide 8-bit data to the data input connections of the main-memory system. Note the $\overline{\text{IOW6}}$ signal connected to the enabling terminal of the block designated ½-Z122 and ½-Z1191. The input or *read* function of Port 6 is tucked away in the $\overline{\text{CSMM}}$ function-select signal. This signal, along with the Port-6 *write* signal, enables the main memory.

All of this means that the main memory ICs are enabled for either kind of Port-6 operation. Data to be stored in the memory (MD0 = MD7) is only available, however, during a Port-6 *write* operation.

And finally, the R/W signal to the main memory ICs determines whether the memory will accept new data or write previously stored data onto the system's main data bus.

These newer 2141 memory chips have been selected for the main memory because they have a nice power-down feature. Whenever they aren't being selected by a logic 0 at CSMM, they automatically power down to a very low level of power dissipation. If it were not for this feature, you would have to add another 5V regulator just to power the main memory functions on a long-term basis.

ADDING MAIN MEMORY TO THE CPU BOARD

All of the components for the main memory system fit onto the existing CPU board. The detailed schematics are shown in Figs. 11-2, 11-3, and 11-4. A suggested parts layout appears in Fig. 11-5 and you will find the parts list for these additional components in Table 11-1.

The circuit in Fig. 11-2 is that of the lower-order operations, consisting of all Port-4 functions as well as a section of Port -6 writing operations. The circuit in Fig. 11-3 practically identical, so it is most important to note the differences before making some wiring

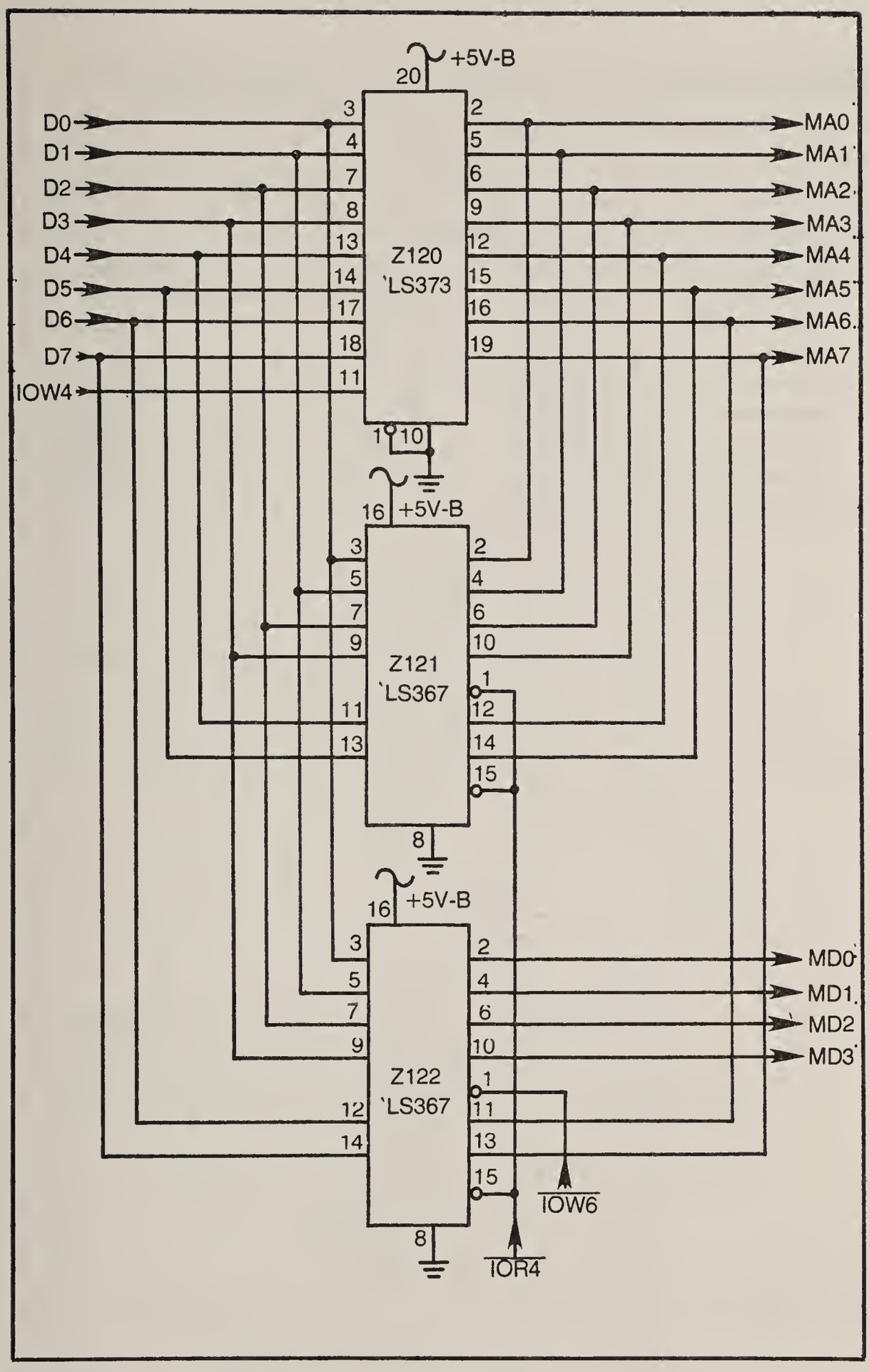


Fig. 11-2. Low-order main-memory schematic diagram.

errors. This circuit handles Port-5 functions and the remainder of the Port-6 writing operation.

The circuit in Fig. 11-3 calls for some special attention. It is only a partial schematic, one showing just two of the eight main-

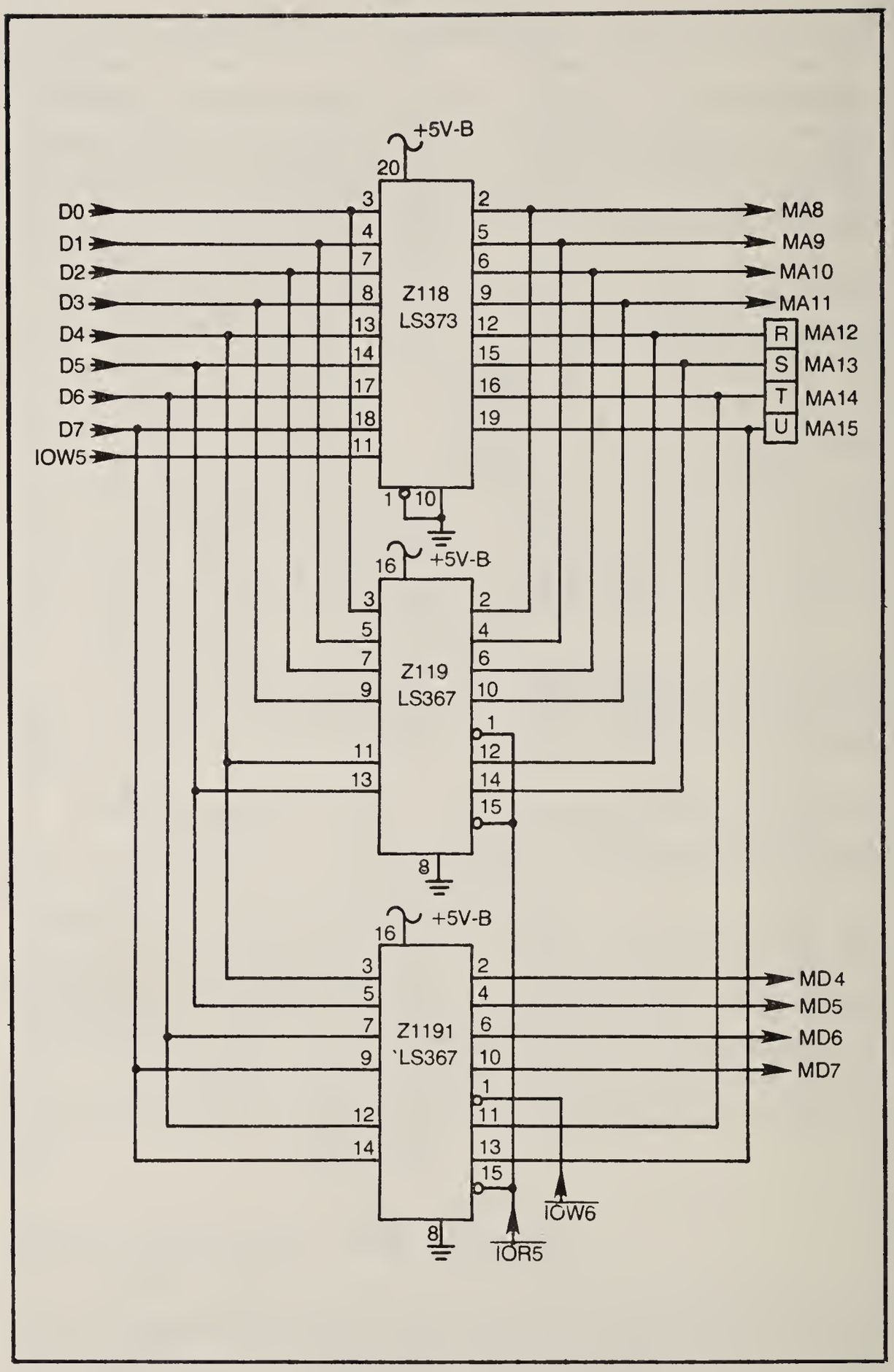


Fig. 11-3. High-order main-memory schematic diagram.

memory RAMs. Wire the two as shown on the schematic (Z123 and Z124), then use the chart in Fig. 11-3 as a guide for wiring the rest of the ICs (Z125 through Z130). As noted on the table, only the connections to pins 7 and 11 are different for each of these eight RAMs.

A 100μF capacitor, C108, appears on the circuit layout diagram in Fig. 11-4, but it does not appear on any of the schematics. I'm sorry, but I did this to catch your attention. The negative terminal of C108 can be soldered to the COMM line along the top of the CPU board. Its positive terminal must go as close as possible to the

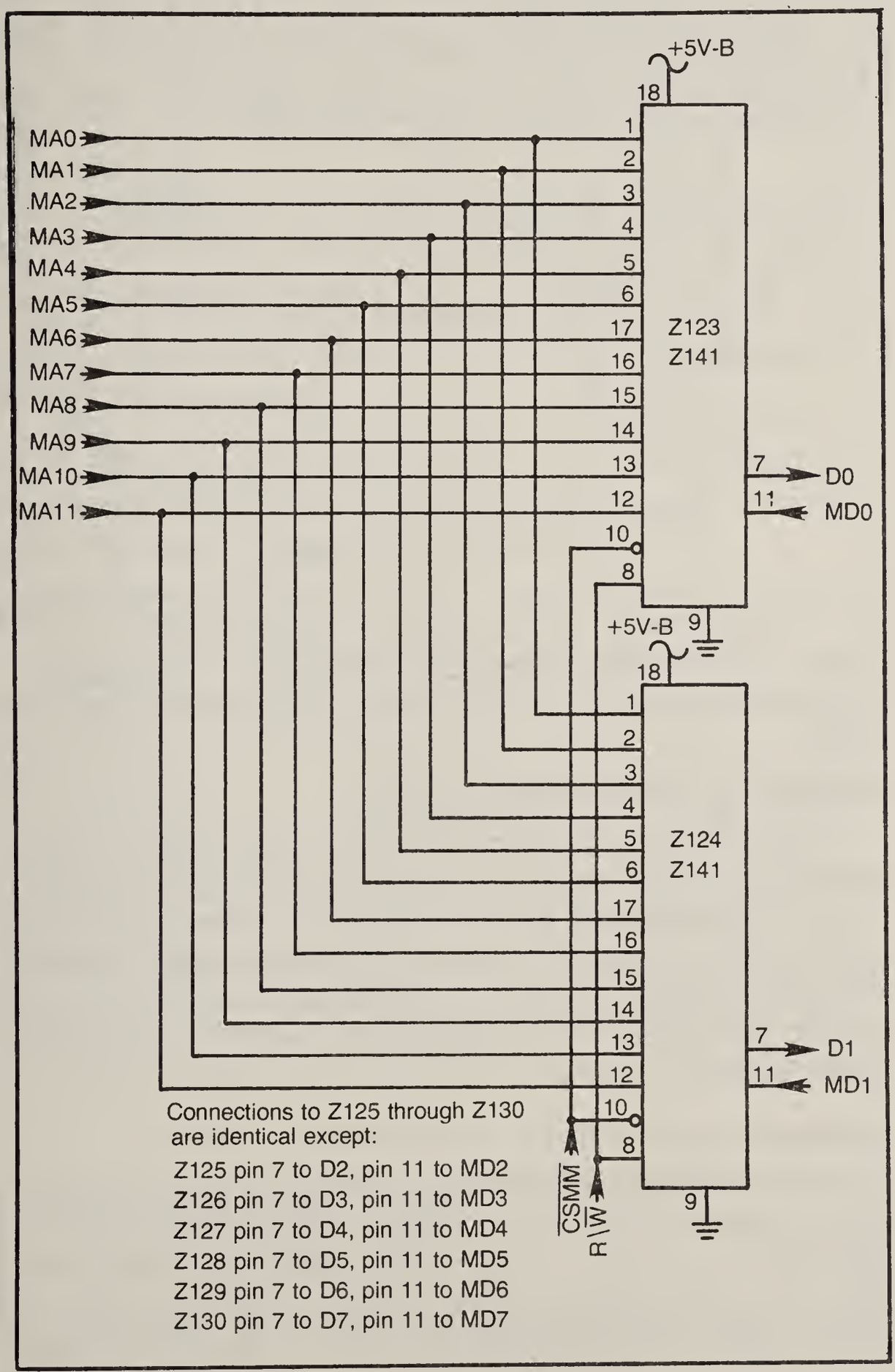


Fig. 11-4. Main memory RAM schematic diagram.

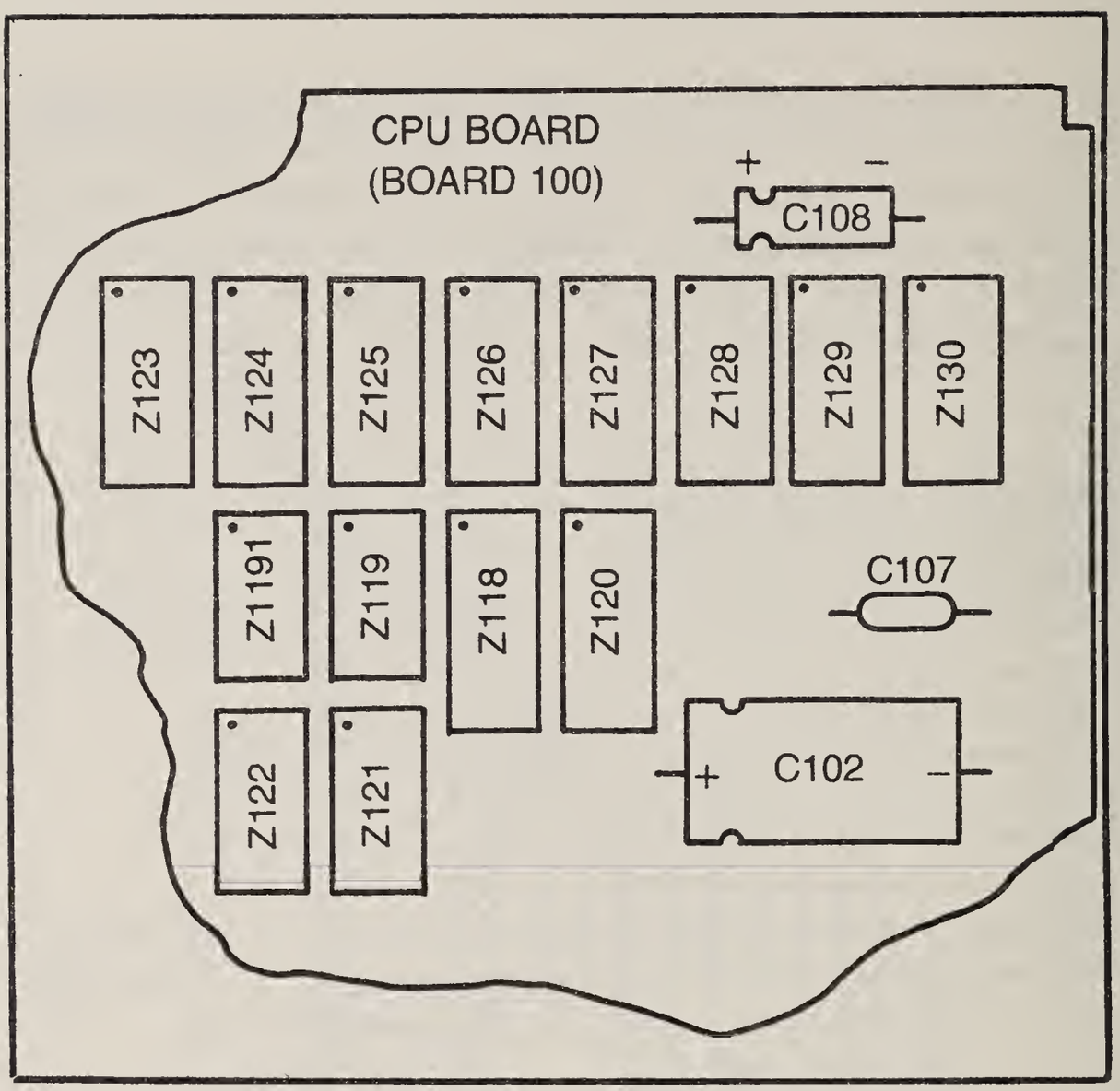


Fig. 11-5. Suggested layout of main memory on the CPU board.

pin-18, +5V-B connection of RAM Z126 or Z127. This capacitor serves as a "bulk decoupler" for the rather extensive main-memory system.

CHECKING THE MAIN-MEMORY SYSTEM

The main-memory system can be tested from either the front panel or by entering a simple diagnostic program into the program memory. The first test is a static test suitable for locating any gross troubles such as poor solder connections, missing wires, and the like. The second test is a dynamic test that confirms the results of the first one and gives you the confidence you need to enter Beta-Class programs.

Front-Panel Tests

Power up the Rodney system, preferably from the auxiliary power supply, RESET the microprocessor, and set the system to its PROGRAM mode. Now, you can take a look at the content of the main memory by setting the panel address switches to the Port-6 address, 806H. The data lamps then indicate the content of the

memory at some address that is not clearly specified at this point. Load a main-memory address such as 0000H. This is going to be a 2-phase operation because it calls for generating a 2-byte address from a 1-byte data bus. Start by addressing Port 4 (804H) on the address switches and setting the data switches to 00H. Momentarily depress the LOAD pushbutton and then note the condition of the data lamps. The data lamps should show 00H, thus confirming proper entry of the 8 lower-order, main-memory address bits.

Now, set the address switches for Port-5 operations (805H), make sure the data switches are still at 00H, and momentarily depress the LOAD pushbutton. Again, the data lamps should confirm a 00H entry.

At this point, the main memory is being addressed with 0000H. Now set the address switches for Port 6 (806H) again. What you see on the data lamps is the contents of the main memory at 0000H. You ought to be able to change this data by simply setting the data switches to some desired configuration and momentarily depressing the LOAD pushbutton.

This is certainly a drawn-out process, and it is certainly not a practical one for checking out the main memory system completely. If everything has worked as described thus far, you can be reasonably certain things are going in the right direction for you.

Some Programmed Tests

Figure 11-6 is the flowchart for a dynamic diagnostic test of the entire main-memory system, including the short-term memory feature. The general idea is to load a MMAL address at Port 4, then read that stored data to see if it matches the original. If it does not, the system lights up all the motor status lamps and goes into a HALT mode. If, on the other hand, there is a good match between the MMAL information which was loaded and then retrieved, a MMAH byte is loaded at Port 5. This information is also checked for a good match and if there is something wrong with the process, the system turns on the motor status lamps and does an automatic HALT. Otherwise, a set of 0s or 1s is loaded into the main-memory data port, Port 6. Like the address ports, this data port is also read and

Table 11-1. Additional parts for the CPU board.

2 ea. 74LS373 8-bit latch (Jameco)
4 ea. 74LS367 hex buffer (Jameco)
8 ea. 2141 4096×1 static RAM
C108, 100μF, 16WVDC electrolytic capacitor (Jameco)

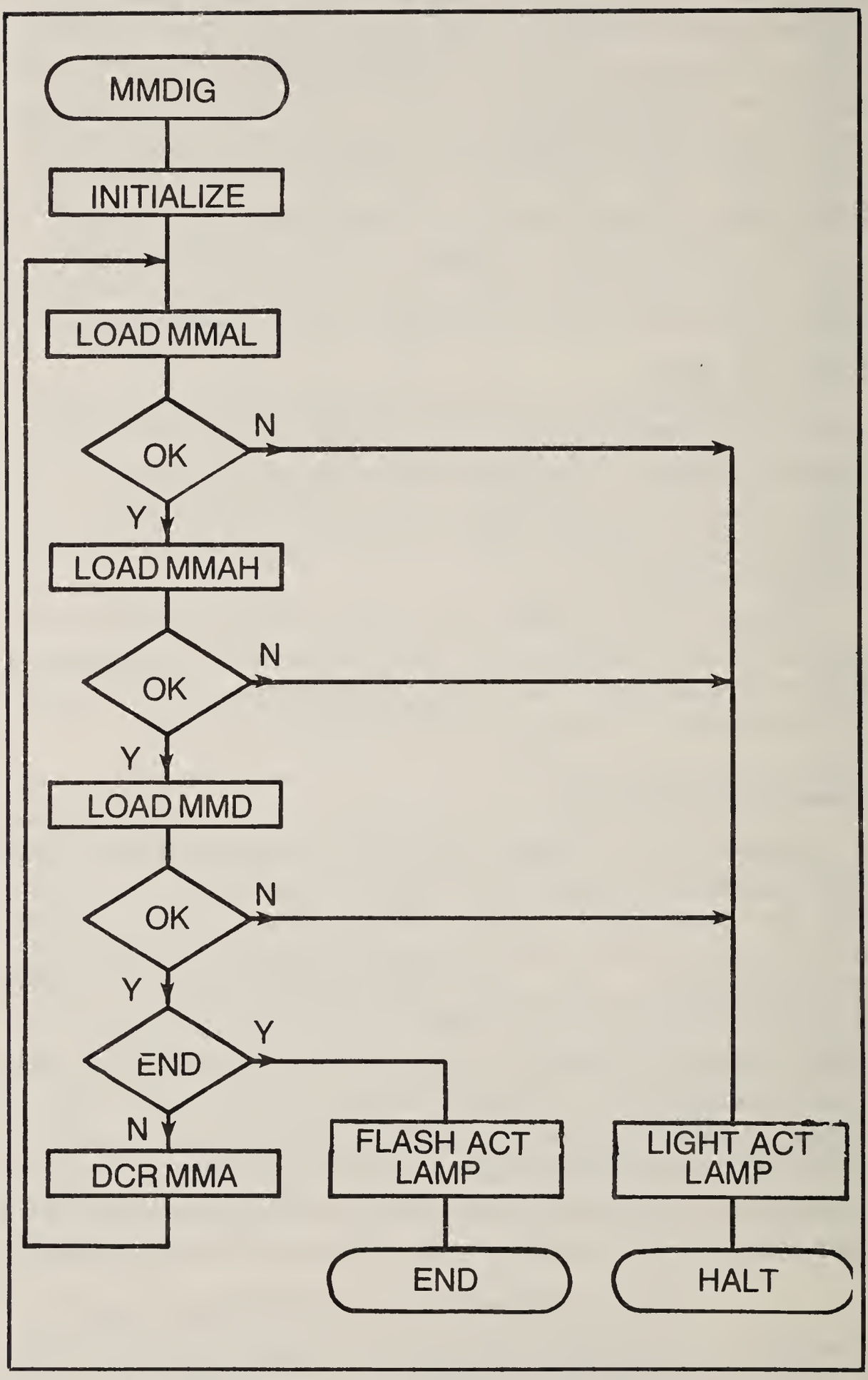


Fig. 11-6. Main memory diagnostic program flowchart.

compared with the original information to see if it loaded and read out properly. A bad match, as far is the data is concerned, throws the system into its HALT mode, as signalled by all motor status lamps turning on.

If the MMAL, MMAH and MMD (main-memory data) all load and read out properly, the system automatically moves on to the next set of addresses. As long as everything goes according to specifications, the system systematically checks all 65k of main memory addressing and data. When the job is completed successfully, the motor status lamps respond by flashing on and off. The lamps continue flashing in this fashion until you either interrupt power to the system or turn on the RESET switch on the front panel.

This process ought to be carried out at least two times; once loading all 0's into the main memory, and then again loading all 1's. The tests should be conducted with the motors switched off at the motor-control panel. Otherwise, Rodney will go through some strange gyrations if anything goes wrong.

Assuming you have loaded the prescribed diagnostic program, you must determine whether you should load 0s or 1s before running the diagnostic. So just before starting to run the program, address program memory location 0200H and LOAD data 00H (for 0s test) or FFH (for 1s test).

After that, you start the test by setting RUN/PROGRAM to RUN and switching off the RESET switch. If all is going well, there will be some time delay before the motor status lamps begin flashing on and off. If and when that happens, you are ready to begin working with Beta Rodney-One. But, if the motor status lamps go on and stay on, there is a problem in the main-memory system that has to be resolved.

In the event of a problem, doublecheck all the wiring and wrap connections, even though the static tests would most likely uncover such troubles. If this visual inspection doesn't uncover the problem, add decoupling capacitors (something between 0.01μF and 0.1μF) between the Vcc and COMM pins of all the main memory IC chips. Sorry, but anyone building up microprocessor memory systems from scratch has to go through this, it all comes with the territory.

Here is the main memory diagnostic program and its subroutines:

```
                    MMDIG:
000  31   3F 03          LXI     SP, TOPS ;SET STACK POINTER
003  21   00 08          LXI     H, PORTO ;
006  36   00             MVI     M,00H    ;TURN OFF MOTOR
                                           STATUS LAMPS
008  21   00 02          LXI     H,TCOD   ;
00B  56                  MOV     D,M      ;FETCH DATA CODE
                                           TO BE TESTED
```

```
00C 01   FF FF        LXI    B,TOPMM ;INITIALIZE MMA POINTER
00F 21   00 04 ;START LXI    H,PORT4 ;INITIALIZE PORT POINTER
002 CD   00 01        CALL   ADRAK
005 C4   50 01        CNZ    NOTOK   ;IF BAD CHECK,
                                     CALL"LIGHT
                                     ;ACT LAMPS" AND HALT
008 CD   00 02        CALL   DECRAD  ;ELSE SET NEXT MMA
00B CC   50 02        CZ     ISOK    ;IF TEST IS DONE, CALL
                                     ;"FLASH ACT LAMPS"
00E C3   0A 00        JMP    START   ;ELSE CYCLE AGAIN
                                     ;END OF MAIN PROGRAM

TOPS EQU 033FH
TCOD EQU 0200H
TOPMM EQU FFFFH
PORT0 EQU 0800H
PORT4 EQU 0804H
PORT5 EQU 0805H
PORT6 EQU 0806H
ORG EQU 0000H
ADRK EQU 0100H
NOTOK EQU 0150H
DECRAD EQU 0200H
ISOK EQU 0250H

              ADRK:
100 71                 MOV    M,C     ;LOAD MMAL AT PORT 4
101 79                 MOV    A,C     ;FETCH MMAL FROM
                                       MMAL POINTER
102 BE                 CMP    M       ;COMPARE MMAL WITH
                                       ORIGINAL
103 C0                 RNZ            ;RETURN WITH NZ IF
                                       NOT SAME
104 2C                 INR    L       ;CHANGE PORT POINTER TO
                                       ;PORT  5
105 70                 MOV    M,B     ;LOAD MMAH AT PORT 5
106 78                 MOV    A,B     ;FETCH MMAH FROM
                                       MMAH POINTER
107 BE                 CMP    M       ;COMPARE MMAH
                                       WITH ORIGINAL
108 C0                 RNZ            ;RETURN WITH NZ
                                       IF NOT SAME
109 2C                 INR    L       ;ELSE ADVANCE PORT
                                       POINTER
                                       ;TO PORT 6
10A 7A                 MOV    A,D     ;FETCH DATA TO BE
                                       LOADED
10B 77                 MOV    M,A     ;LOAD DATA AT
                                       PORT 6
10C BE                 CMP    M       ; COMPARE MMD WITH
                                       ORIGINAL
10D C0                 RNZ            ;RETURN WITH NZ
                                       IF NOT SAME
10E C9                 RET            ;ELSE RETURN WITH Z
              NTOK:
150 21   00 08        LXI    H,PORTO ;POINT TO PORT 0
153 36   0F           MVI    M, OFH  ;SET FOR "LIGHT ACT
```

```
                                                    ;"LAMPS"
155  76                        HLT                  ;HALT THE TEST UNTIL
                                                    ;MANUALLY RESET

                          ISOK:
250  AF                        XRA       A          ;CLEAR REGISTER A
251  21   00 08                LXI       H, PORTO   ;POINT TO PORT 0
254  1E   08          ;SETE    MVI       E,08H      ;INITIALIZE E COUNTER
256  06   FF          ;SETB    MVI       B,FFH      ;INITIALIZE B COUNTER
258  0E   FF          ;SETC    MVI       C,FFH      ;INITIALIZE C COUNTER
25A  0D               ;DECC    DCR       C          ;DECREMENT C COUNTER
25B  C2   5A 02                JNZ       DECC       ;IF NOT ZERO,DECREMENT
                                                    ;AGAIN
25F  05                        DCR       B          ;ELSE DECREMENT B COUNTER
260  C2   58 02                JNZ       SETC       ;IF  NOTZERO, RESET
                                                    ;C COUNTER AND DO AGAIN
263  1D                        DCR       E          ;ELSE DECREMENT E COUNTER
264  C2   56 02                JNZ       SETB       ;IF NOT ZERO, RESET
                                                    ;B COUNTER AND DO AGAIN
267  2F                        CMA                  ;ELSE COMPLEMENT A
268  77                        MOV       M,A        ;CHANGE LAMP STATUS
269  C3   54 02                JMP       SETE       ;AND  REPEAT ENTIRE CYCLE
                                                    ;END OF ISOK
                          DECRAD:
200  0D                        DCR       C          ;DECREMENT MMAL POINTER
201  C0                        RNZ                  ;IF NOT ZERO,
                                                    RETURN WITH
                                                    ;NZ
202  05                        DCR       B          ;ELSE DECREMENT MMAH
                                                    ;POINTER
203  C8                        RZ                   ;IF ZERO, RETURN
                                                    WITH Z
204  06   FF                   MVI       B,FFH      ;ELSE RESET MMAL
                                                    POINTER
206  C9                        RET                  ;AND RETURN WITH NZ
                                                    ;END OF DECRAD
```

This completes the listing of the main memory diagnostic program. Good luck.

RUNNING BETA RODNEY

The essence of the Alpha-Class version of machine intelligence is its purely reflexive and largely random behavior. Beta-Class machines implement the primary features of the Alpha-Class versions, but extend them to include remembering responses, presumably responses that were most successful in the past.

The main-memory circuitry you have just assembled and tested gives Rodney the basic equipment he needs for functioning as a more efficient machine. The thing to watch for in this mode of behavior is how effective his work becomes with increasing levels of experience. He will start out acting just like Alpha Rodney-One but as knowledge accumulates in his main memory, the machine will spend far less time dealing with stall situations and a great deal more time

running freely around the room.

The Rodney Beta-One software described in this chapter does not include any special provisions that force Rodney to actively seek out his battery charger. Like the Alpha version, he will find it only by accident. *But* (and this is an important point) you will find Beta Rodney is far more effective at finding the battery charger. Why is that? Because he spends more time criss-crossing the room, and the more time he spends running freely, the greater the chances of stumbling across the battery charger. Beta Rodney, in other words, is a more efficient creature in terms of caring for his own survival. He is a higher-order creature on the evolutionary scale of machine intelligence.

Beta Rodney is *not* specifically programmed to find his battery charger any better than Alpha Rodney—note this statement very carefully. The addition of some memory makes Beta Rodney run more effectively, and by running more effectively, he has a better chance of finding the battery charger in a shorter period of time.

So the Beta Rodney-One programming presented here will give your machine a greater amount of effectiveness when it comes to survival. Bear than in mind while loading this rather lengthy and sometimes tedious program.

The flowchart in Fig. 11-7 represents an overall view of the Beta Rodney-One program. As in the case of the simpler Alpha program in Chapter 10, it is important to understand the basic mode of behavior and some definitions are called for before anything else.

Definitions of Special Terms and Acronyms

INITIALIZE—A housekeeping operation which initializes the microprocessor's stack pointer to the top of the program RAM.

CLEAR MM—This operation clears the contents of the main memory to zeros. Rodney thus begins with absolutely no knowledge concerning the ways of the world and how to deal with it. If the main memory were not completely erased in this fashion, it would be full of random garbage. Rodney would certainly respond to this garbage, but isn't it always better to start a project knowing nothing at all than to begin with your head full of a lot of misinformation?

RUN ALPHA-PLEX—This is a rather extensive subroutine that includes most of the features of an Alpha-Class machine. Virtually all of the operations in RUN ALPHA-PLEX come directly from Alpha Rodney-One, but it serves two different purposes at two different places in this program. It is used first for determining whether or not a stall condition exists, and then it is used at a later time to determine whether or not a selected response is indeed

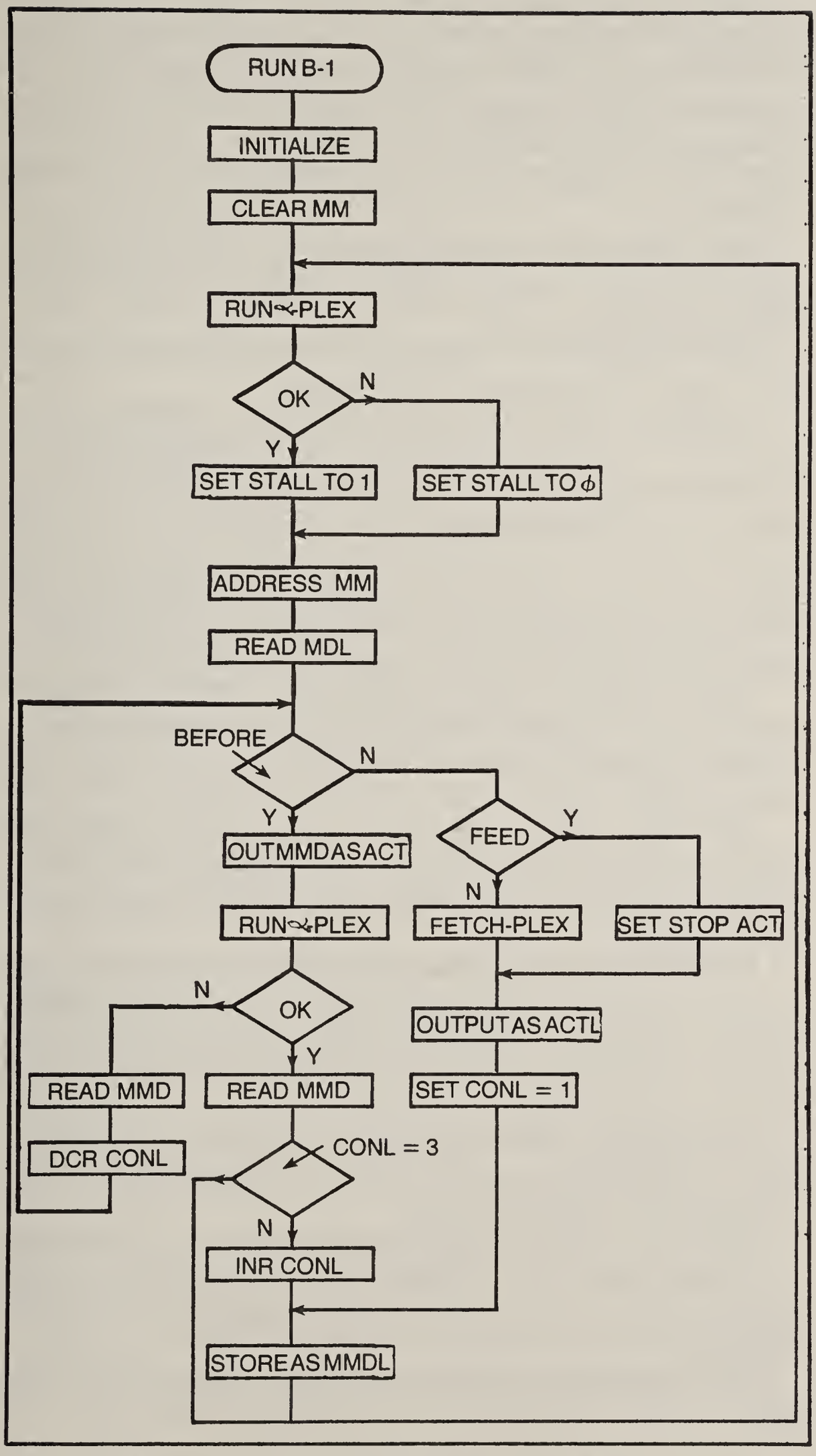


Fig. 11-7. Flowchart for the Beta Rodney-One main program.

working.

OK—The OK operation always follows RUN ALPHA-PLEX and serves a twofold purpose. In the first instance, an N output from OK indicates the machine is stalled, while a Y output signals a not-stalled, or "ok" condition. OK functions the same way later in the program, but this time an N output indicates the selected response is not working, while the Y output indicates the response is a workable one.

SET STALL TO 1—Here the stall bits in ENVL (bits D4 and D5) are actively set to logic 1, thereby acting as a not-stall flag for further operations.

SET STALL TO 0—This operation sets the stall bits of the ENVL word to 0, indicating a stall condition.

ADDRESS MM—The ENVL code, as modified by SET STALL TO 1 or SET STALL TO 0, is used for addressing the main-memory system. This is the ADDRESS MM operation. The main memory is always addressed with an ENVL code which is picked up during the first RUN ALPHA-PLEX. In this particular version of the Beta programming, Rodney will be using only the lower-order environment byte, ENVL. Thus, the main-memory addressing involves only the lower-order address, MMAL of Port 4.

READ MMDL—"Read the main-memory data" from Port 6. The idea here is to first find out whether or not the machine has ever encountered this ENVL condition before and, if so, give the previous response a good try.

BEFORE—"Have I ever encountered this exact situation before?" The answer to the question is implied by the status of two confidence-level bits in the MMD word which was fetched during READ MMDL.

OUT MMD AS ACT—"Output the stored action from the past as an ACTL solution to the present situation." The system in other words, is instructed to do whatever had been done before in the present ENVL situation. This is the point where the main-memory scheme really pays off.

DCR CONL—This is a step that is responsible for decreasing Rodney's confidence in a particular response. In plain text it means, "Decrement the confidence level."

CONL=3—"Is the confidence level of a remembered response at its highest level," *e.g.*, decimal 3? There is no advantage in using confidence levels any greater than 3 for any higher-order machine intelligence.

INR CONL—Here Rodney's confidence is increased by one unit.

STORE AS MMDL—A response that "works" and its associated 2-bit confidence level are stored as data in the main memory. The address of this data is, of course, predetermined by the ENVL word that existed at the time Rodney was called upon to make a response. STORE AS MMDL loads the response information into the main memory at a location determined by the environmental condition that existed a moment earlier. Here is where the system's short-term memory is demonstrated.

FETCH-PLEX—This is a subroutine that is taken directly from Alpha Rodney-One. It is responsible for selecting a random response code and checking to make certain it represents a valid motion code (as opposed to one that would make Rodney stop in his tracks).

OUTPUT AS ACTL—The random response from FETCH-PLEX is loaded as an ACTL to Port 0.

SET CONL=1—The confidence level bits accompanying a randomly generated motion code are set to decimal 1.

The Main Flowchart Analysis

After studying the definitions just outlined, you should already have some good notions about how this Beta Rodney-One scheme works. Nevertheless, a brief analysis is in order.

The program starts by initializing the stack pointer and clearing the main memory. These are both one-time operations which merely serve to get things started on the right foot (or wheel as this case might demand).

The first operational step is a RUN ALPHA-PLEX routine, followed by a conditional OK. In this case, RUN ALPHA-PLEX fetches the ENVL code and checks it for stall and feeding conditions. The OK operation then determines whether the results call for a stall-type response or a go-ahead response.

If the system is both stalled AND feeding, or it is NOT stalled, the response from OK is Y and the stall bits in the ENVL code are actively set to 1s. Otherwise, they are set to 0s.

We have no way of knowing what Beta Rodney will do when this program is first started. It depends on what motion code happens to come up at ACTL when power is turned on, and that isn't specified anywhere. There is a 1:4 chance ACTL will come up with a stop code, but that's all right, the RUN ALPHA-PLEX operation will get things moving within a few milliseconds.

There's no telling what will be entered into the system at RUN ALPHA-PLEX. To be sure, the system will respond anyway. The ENVL code, whatever it might be, is then loaded into MMAL, Port

4, at the ADDRESS MM step. The condition in the environment is, itself, the address for the main memory.

The next step is to read the contents of the memory at that location, the READ MMDL step. Now, if the system is just starting up, the data will be all 0s as dictated by the CLEAR MM step. But, on the other hand, if the system has been working for some time and Rodney has encountered this particular ENVL before, there will be some sort of action code and confidence-level information in the MMDL data.

After reading MMDL, the system first checks the confidence level bits, D6 and D7 of MMDL. These two bits actually comprise a 2-bit binary number that can have decimal equivalents between 0 and 3. Zero represents the lowest possible confidence level and is a sure sign that Rodney has never encountered this particular ENVL situation before.

So the BEFORE question really wants to know if the confidence level is zero or not. If the confidence level is zero, the response is N, and the system switches over to pick up a random response. And why not? Rodney doesn't have the foggiest notion about what to do—so he simply picks out something at random. See how the Alpha features buttress the Beta-Class operations?

However, let's see what happens if the confidence level happens to be greater than zero, a clear indication that Rodney has encountered this particular ENVL situation before. If indeed he has encountered the situation before, the next step is to try out the response he made before. This is the OUT MMD AS ACT step. The four lower-order bits in the MMDL represent the previous action, and it is immediately loaded as ACTL at Port 0.

The question at this point is: Does the old response still work? Now one might think the answer should be a certain yes, but this isn't necessarily the case. Rodney is working in a real world, you see, and the real world has a way of pulling some tricks on all of us. The complexity of the real world is virtually infinite compared to the simple range of responses and senses Rodney has available to him. Slight changes in angles and velocities from one sort of contact to another can be meaningful; just because Rodney's rather coarse view of things resulted in a workable response in the past doesn't necessarily mean the same response will work again, particularly since Rodney's coarse view will most certainly miss a few subtleties now and then.

This all a rather critical notion in the evolution of machine intelligence as it is presented in this book. It is a far-reaching notion that calls for some thoughtful consideration on the part of anyone

expecting to do some serious work with this particular brand of machine intelligence. It is something that separates Rodney from the mainstream approaches to artificial intelligence and gives him a high level of adaptability that totally escapes traditional machine programming philosophies. Sorry we have to leave it at that.

So let's suppose the answer to OK is "N" for some reason. The system got into this loop only by having encountered the situation before, and that means the confidence level has to be something greater than zero. But if the response doesn't work, the confidence level is decremented one count by DCR CONL.

Control from DCR CONL is then looped back to BEFORE. Now here's an important point—if the confidence level is still greater than zero after DCR CONL, the answer to BEFORE is also still "Y" and *Rodney tries the same response again*. If it doesn't work a second time, the answer to OK is still "N" and the confidence level is decremented once again by DCR CONL before looping back up to BEFORE.

It is quite possible that Rodney might enter this loop with a confidence level of 3—the highest possible level. If the previous, high-confidence response doesn't work, Rodney gets the dickens knocked out of him each time the response is tried. In other words, Rodney is going to keep trying a high-confidence response until his confidence is reduced to zero. If the response worked four times before, shouldn't it work again? But if it doesn't work, as determined by looping around through DCR CONL several times, it wasn't the best possible response anyway and it ought to be eliminated. That's what happens—Less-than-best responses are eliminated and eventually replaced with better ones. *Rodney's knowledge is refineable* and that is one of the key elements of a Beta-Class machine.

We could really get wrapped up in a lot of detail and machine-intelligence theory at this point, but we have to leave this most intriguing loop for the sake of getting the machine working. For the sake of those who are following the real meat of this matter, I'd like to point out that the RUN ALPHA-PLEX step includes at least one FETCH ENVL update. Rodney's persistence at trying an action that worked before just might alter the environment in such a way that a whole new set of meaningful circumstances arise—the relevant, but finer points of a given situation might come to the surface.

There are two ways out of the DCR CONL loop, either running the confidence level down to zero or by finding that the old response does indeed work. In the first case, Rodney gets out of the loop by the "N" path from BEFORE, and he goes over to pick up a new random response—"What worked before doesn't work now, so I'll

try something new". In the second case, Rodney leaves the loop via the "Y" response of OK. This would mean that the old response works. Now, if the confidence level of the old response is not already at the highest point, the "N" output of CONL-3 goes to INR CONL where the confidence level is incremented by one. Otherwise the confidence level is left at 3.

In either case, the successful response and the confidence level bits are stored in the main memory as data. This information is placed at an address representing the ENVL condition that prompted the action in the first place. After depositing this information for future reference, control loops back to the first RUN ALPHA-PLEX where the whole thing starts over again.

A negative reply to BEFORE actually indicates the machine has no response available, the confidence level is zero, either because Rodney has never encountered the situation or has not solved the problem with any success. In any event, the program calls the FETCH-PLEX subroutine where a random response is selected and checked for validity (something other than a stop code). This random response is loaded in Port 0 as ACTL, the confidence level is set to 1, and the results are stored as data in the main memory.

Experimenters who are skilled at reading flowcharts and relating them to machine activity are going to have some problems understanding how this system can possibly work as described. The problem is that I haven't completely described the full spectrum of Beta-Class behavior. In a traditional sense, the scheme would seem to create a hopeless mess of Rodney's behavior. He seems to go into a series of meaningless, random spasms that leave him shuddering helplessly in the middle of the floor.

Indeed Rodney will seem to be completely crazy at first, but within a few moments, even some of the spasms will take on higher confidence levels which will cause things to settle down gradually. There is a good chance that Rodney will not stop the first time he encounters the battery charger. He won't stop because he doesn't know he is supposed to do so. Whatever initial response he makes to the battery charger will be remembered, however, and the second time he finds it, he will see that a response (other than stopping) is not the correct one. Our "magic loop" through DCR CONL takes care of that.

BETA RODNEY-ONE PROGRAMS

Study the following programs, comparing them with their corresponding flowcharts. A complete study of the programming steps at this time will eliminate any unfortunate surprises that might

otherwise crop up when actually entering them into the system.

Main Program

This program follows the flowchart for the main Beta Rodney-One sequence in Fig. 11-7. The numerous NOP (no operation) commands are inserted to allow for additional Gamma-Class commands in Chapter 12.

```
                                        BETA1:
000  31   FF 03          LXI     SP,TOPS  ;SET STACK POINTER
                                           TO TOP
003  CD   00 01          CALL    CLRM     ;"CLEAR MM"
006  21   00 08          LXI     H,PORT0  ;SET PORT POINTER TO
                                           ;PORT 0
009  CD   50 01 ;FETEN   CALL    RUAPX
00C  7E                  MOV     A,M      ;FETCH ENVL
00D  C2   15 00          JNZ     SSB1     ;IF "OK" IS Y,
                                           JUMP TO
                                           ;"SET STALL TO 1"
010  E6   CF            ANI     CFH      ;ELSE SET "STALL
                                           BITS TO 0"
012  C3   17 00          JMP     AMM      ;AND JUMP TO "ADDRESS
                                           MM"
015  F6   30    ;SSB1    ORI     30H      ;SET STALL BITS TO 1
017  32   04 08 ;AMM     STA     PORT4
01A  3A   06 08 ;SAVD    LDA     PORT6
01D  47                  MOV     B,A      ;SAVE MMD IN B
01E  E6   C0             ANI     COH      ;ISOLATE CONL BITS
020  CA   5C 00          JZ      FEDUP    ;IF "BEFORE" IS N, JUMP
                                           ;TO "FEED"
023  78                  MOV     A,B      ;ELSE FETCH MMD FROM B
024  F6   F0             ORI     FOH      ;CONDITION ACTL
026  77                  MOV     M,A      ;AND "OUT MMD AS ACT"
027  CD   50 01          CALL    RUAPX    ;CALL "RUN ALPHA-PLEX"
02A  CA   4D 00          JZ      RMMD     ;IF "OK" IS N, JUMP TO
                                           ;"READ MMD"
02D  3A   06 08          LDA     PORT6    ;ELSE FETCH MMD
030  E6   C0             ANI     COH      ;ISOLATE CONL BITS
032  FE   C0             CPI     C0H      ;"CONL=3"—Z IF YES,
                                           ;NZ IF NO
034  CA   09 00          JZ      FETEN    ;IF YES, LOOP BACK
                                           TO START
                                           ;OF CYCLE
037  C6   40             ADI     40H      ;ELSE "INR CONL"
039  4F                  MOV     C,A      ;SAVE CONL IN C
03A  3A   06 08          LDA     PORT 6   ;FETCH MMD
03D  E6   3F             ANI     3FH      ;SET CONL BITS TO ZERO
03F  B1                  ORA     C        ;COMPOSE NEW MMD WORD
040  32   06 08 ;SAMMD   STA     PORT.    ;"STORE AS MMDL"
043  00                  NOP
044  00                  NOP
045  00                  NOP
046  00                  NOP
047  00                  NOP
048  00                  NOP
049  00                  NOP
```

```
04A  C3   09 00          JMP     FETEN   ;LOOP BACK TO START
                                           OF CYCLE
04D  3A   06 08 ;RMMD     LDA     PORT6   ;FETCH MMD
050  D6   40             SUI     40H     ;"DOR CONL"
052  00                   NOP
053  00                   NOP
054  00                   NOP
055  00                   NOP
056  00                   NOP
057  00                   NOP
058  00                   NOP
059  C3   1A 00          JMP     SAVD    ;JUMP BACK TO "BEFORE"
05C  3A   04 08 ;FEDUP    LDA     PORT4   ;FETCH MMA/ENVL
05F  E6   80             ANI     80H     ;ISOLATE FEED BIT
                                         ;Z IF YES, NZ IF NO
061  C2   68 00          JNZ     FETPL   ;IF NO, JUMP TO
                                           "FETCH -
                                         ;PLEX"
064  AF                   XRA     A       ;CLEAR ACCUMULATOR
065  C3   6B 00          JMP     OUACT   ;JUMP TO "OUTPUT AS
                                           ACTL"
068  CD   20 10 ;FETPL    CALL    FETR
06B  77      ;OUACT      MOV     M,A     ;"OUTPUT AS ACTL"
07C  F6   80             ORI     80H     ;"SET CONL=1"
07E  C3   40 00          JMP     SAMMD   ;JUMP TO "STORE
                                           AS MMDL"
                                         ;END OF BETA1

TOPS  EQU 033FH; PORTO  EQU  0800H; PORT4  EQU  0804H; FETR  EQU  01
PORT6  EQU  0806H;  CLRM  EQU  0100H;  RUAPX  EQU  0150H
```

CLRM Subroutine

This subroutine is responsible for initially clearing the main memory of all data, *e.g.*, setting all data to zeros.

```
                  CLRM:
100  06   FF             MVI     B, MMALT ;SET MMA POINTER TO
                                           ;TOP ADDRESS
102  AF                   XRA     A       ;CLEAR A TO ALL ZEROS
103  21   04 08          LXI     H,PORT4 ;SET PORT POINTER TO
                                           ;MMAL
106  70      ;NEWAD      MOV     M,B
107  21   06 08          LXI     H,PORT6 ;SET PORT POINTER TO
                                           ;MMDL
10A  77                   MOV     M,A     ;MOVE 0'S TO MMD
10B  05                   DCR     B       ;DECREMENT MMA POINTER
10C  C2   06 01          JNZ     NEWAD   ;IF NOT ZERO, DECREMENT
                                           ;TO A NEW ADDRESS
10F  77                   MOV     M,A     ;ELSE CLEAR BOTTOM MMA
110  C9                   RET             ;RETURN TO MAIN PROGRAM
                                           ;END TO CLRM

MMALT    EQU     FFH
```

FETR and TIME 1 Subroutines

Although the specified addresses are different, the two subroutines that follow are nearly identical to those used in the Alpha

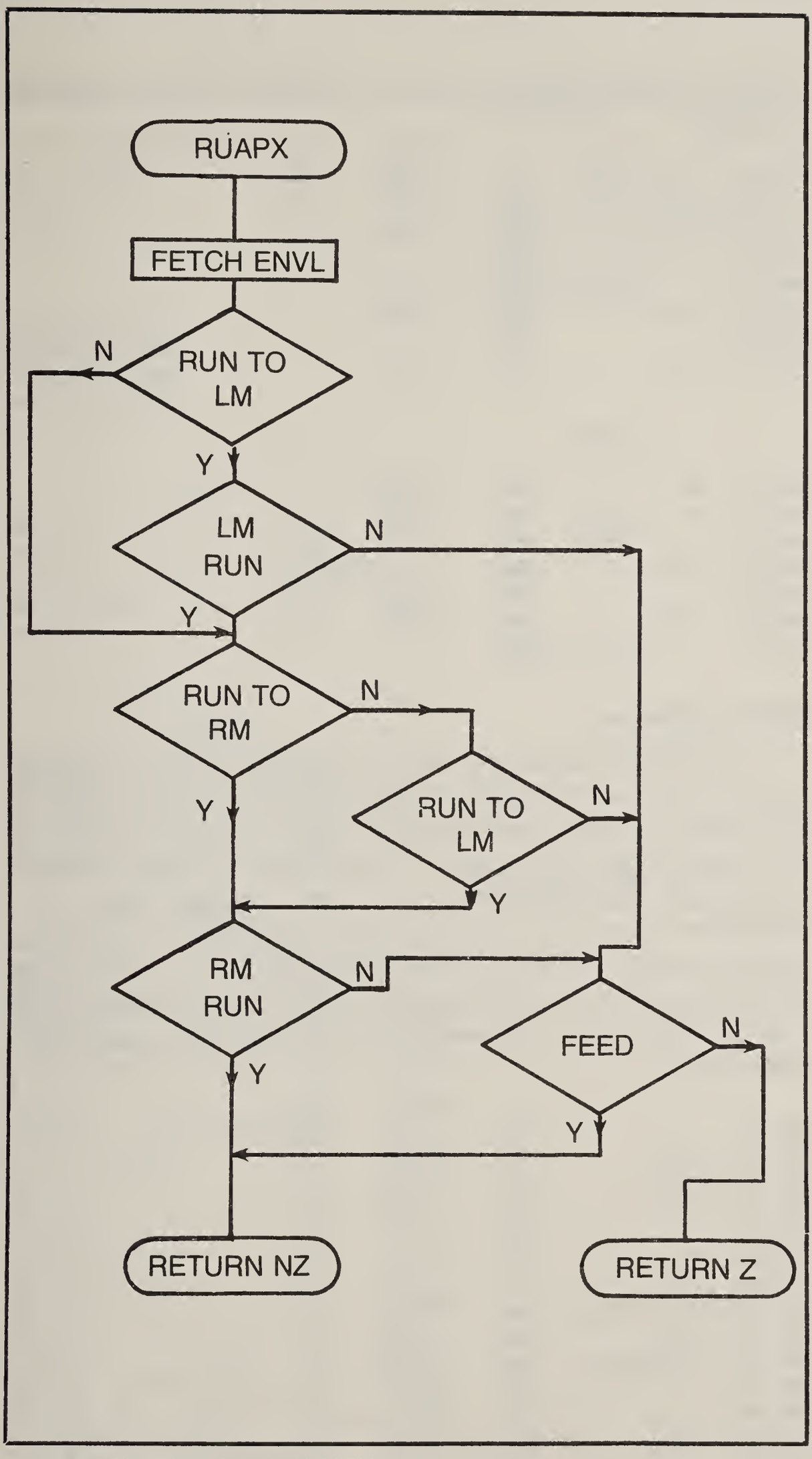


Fig. 11-8. Flowchart for the RUAPX subroutines.

system of Chapter 10. They are thus presented without additional comments.

```
                        FETR:
120  3A   02 08  ;STRT    LDA     TSWR
123  4F                   MOV     C,A
124  CD   00 02           CALL    RTLM
127  C0                   RNZ
128  79                   MOV     A,C
129  CD   25 02           CALL    RTRM
12C  C0                   RNZ
12D  C3   20 01           JMP     STRT


                    TIME1:

3A0  06   FF              MVI     B,FFH
3A2  1E   FF      ;SETE   MVI     E, FFH
3A4  1D           ;DCRE   DCR     E
3A5  C2   A4 03           JMP          DCRE
3A8  05                   DCR     B
3A9  C2   A2 03           JNZ     SETE
3AC  C9                   RET
```

RUAPX Subroutine

The RUAPX subroutine for Beta Rodney-One is a modified version of the master program for Alpha Rodney-One. Since many of the details have been painstakingly described in Chapter 10, there is little need for further commentary on the flowchart in Fig. 11-8 and its corresponding subroutines. You might note how deeply the subroutines become nested in this scheme. There are subroutines calling subroutines that call other subroutines. It's a good thing Rodney has a built-in standby power system, hours of programming could be completely wiped out with a utility power failure lasting only milliseconds.

```
                              RUAPX:
150  7E                   MOV     A,M       ;FETCH NEW ENVL
151  CD   00 02           CALL    RTLM
154  CA   60 01           JZ      RURM
157  CD   50 02           CALL    LMRN
15A  79                   MOV     A,C
15B  E6   10              ANI     10H
15D  CA   70 01           JZ      FEED
160  CD   25 02  ;RURM    CALL    RTRM
163  C2   74 01           JNZ     RULM2
166  CD   75 02  ;RMN     CALL    RMRN
169  79                   MOV     A,C
16A  E6   10              ANI     10H
16C  CA   70 01           JZ      FEED
16F  C9                   RET               ;RETURN WITH Y AS NZ
170  7E           ;FEED   MOV     A,M
171  E6   40              ANI     40H
```

```
173  C9                        RET             ;RETURN WITH Y AS NZ
                                               ;OR N AS Z
174  CD   00  02  ;RULM2       CALL    RTLM
177  C2   66  01               JNZ     RMN
17A  C3   70  01               JMP     FEED
17D  C7                        RST     0       ;RESET ENTIRE PROGRAM
                                                   IF
                                               ;THINGS BLOW UP
                                               ;END OF RUAPX

RTLM EQU 0200H
RTRM EQU 0225H
LMRN EQU 0250H
RMRN EQU 0275H



                   RTLM:
200  47                        MOV     B,A
201  87                        ADD     A
202  A8                        XRA     B
203  E6   02                   ANI     02H
205  C9                        RET
                                               ;END OF RTLM


                   RTRM:
225  47                        MOV     B,A
226  87                        ADD     A
227  A8                        XRA     B
228  E6   08                   ANI     08H
230  C9                        RET
                                               ;END OF RTRM
                   LMRN:
250  7E                        MOV     A,M
251  E6   EF                   ANI     EFH
253  77                        MOV     M,A
254  0E   04                   MVI     C,04H
256  16   08                   MVI     D,08H
258  CD   A0  03  ;CTIM        CALL    TIME1
25B  7E                        MOV     A,M
25C  A9                        XRA     C
25D  E6   10                   ANI     10H
25F  CA   6B  02               JZ      DCONT
262  A9                        XRA     C
263  4F                        MOV     C,A
264  E6   0F                   ANI     0FH
266  3D                        DCR     A
267  CA   76  02               JZ      SETOK
26A  0D                        DCR     C
26B  15           ;DCONT       DCR     D
26C  C2   58  02               JNZ     CTIM
26F  0E   00      ;SNOK        MVI     C,00H
271  7E           ;TOFL        MOV     A,M
272  F6   F0                   ORI     F0H
274  77                        MOV     M,A
275  C9                        RET
276  0E   10      ;SETOK       MVI     C,10H
```

```
278  C3  71 02          JMP     TOFL
                                        ;END OF LMRN
                RMRN
275  7E             MOV     A,M
276  E6  DF         ANI     DFH
278  77             MOV     M,A
279  0E  04         MVI     C,04H
27B  16  08         MVI     D,08H
27D  CD  A0 03 ;CTIM2  CALL    TIME1
280  7E             MOV     A,M
281  A9             XRA     C
282  E6  20         ANI     20H
284  CA  90 02      JZ      DCONT2
287  A9             XRA     C
288  4F             MOV     C,A
289  E6  0F         ANI     0FH
28B  3D             DCR     A
28C  CA  9B 02      JZ      SETOK2
28F  0D             DCR     C
290  15        ;DCONT2  DCR     D
291  C2  7D 02      JNZ     CTIM2
294  0E  00    ;SNOK2   MVI     C,00H
296  7E        ;TOFL2   MOV     A,M
297  F6  F0         ORI     F0H
299  77             MOV     M,A
29A  C9             RET
29B  0E  10    ;SETOK2  MVI     C,10H
29D  C3  96 02      JMP     TOFL2
                                        ;END OF RMRN
```

An Elementary Gamma Rodney

So now it's Gamma Rodney. Starting to work on a Gamma-Class machine is something like opening the door to a strange, dark room; you are never quite sure what you are going to find in that room. Whatever is there might turn out to be very exciting or, on the other hand, it might be rather plain and uninspiring. Whether the whole thing is exciting or not depends a lot on your own perspective.

From an experimenter's point of view, Gamma-Class schemes are intriguing because there are so many different ways to implement them. Even in the rather restrictive context of a motion-expressing machine such as Rodney, there are at least a dozen different ways to approach the Gamma-Class mode of operation. The Gamma scheme presented in this chapter represents perhaps the simplest that is possible with the hardware available.

There is no real problem coming up with a scheme that satisfies the definition of a Gamma-Class machine. Just devise a system capable of sensing responses that have a high confidence level and generalizing these responses to similar environmental situations that carry lower confidence levels.

The problem is coming up with a Gamma-Class scheme that imposes the fewest restrictions upon the machine's behavior. A Gamma-Class machine must be able to detect elements of the response that are most relevant to a given set of environmental conditions. And, that's no easy task, even for a wonderful, gee-whiz microprocessor. Here's the real question, though: How does the machine determine what is relevant and what is not?

For the sake of getting a Gamma-Class machine rolling around the floor, we are going to settle for some definitions of "relevance" that have to be imposed by the program itself. One day we can hope to let the machine set its own procedures and standards for "relevance." This will be a matter of letting a secondary Gamma-Class mechanism build up the definitions for a primary Gamma scheme that

interacts more directly with the environment. At the present time, however, we are going to settle for some definitions of "relevance" that are dictated by a subroutine dubbed RELSET—*RELevance SET.*

Here is yet another unusual feature of Gamma Rodney and Gamma-Class machine behavior in general. It is not easy to determine whether or not the Gamma mechanism is really working. On the surface of things, your Gamma-Class Rodney machine won't appear to behave differently from the Beta version. You must study the machine carefully to detect the differences, but once you recognize them, you can begin to appreciate their true significance. In fact, if you don't know what to look for, implementing the Gamma program will appear to be a great waste of time and effort. Thus, the matter must be put into some perspective from the outset.

HOW TO APPRECIATE GAMMA-CLASS BEHAVIOR

Rodney is an adaptive machine. He doesn't *do* anything remotely practical, and other than stopping at his battery charger every once in a while, his actions seem to be entirely pointless. Rodney's purpose in life is to demonstrate the adaptive behavior of a certain kind of machine intelligence. Ultimately, we can show that adaptive behavior is paramount to implementing machine intelligence on a more practical level. Without the ability to adapt to changing environmental conditions, any robot machine can be quickly reduced to a shuddering mass of hardware which is apt to do more harm than good.

If you have been studying the Alpha and Beta versions of your Rodney machine carefully, you have already noted how much more efficiently the Beta version functions. The Beta-Class machine spends far less time blundering around corners and obstacles than the more primitive Alpha-Class machine does.

My own work with Rodney and a computerized graphics simulator shows some interesting contrasts between Alpha- and Beta-Class behavior. For one thing, an Alpha-Class machine scores fairly close to *chance* when it comes to the ratio of "good responses" to the total number of encounters with obstacles. The chance figure is not 50-50 because the number of "good" responses in a corner-trap situation is less than an encounter with a flat wall or small immovable object. The chance figure depends on the range of responses available to the machine and the character of the environment. However, for the sake of argument, suppose the ratio of "good" responses to total encounters is 0.64.

Now this particular Alpha-Class machine, the one with a score of 0.64, will maintain the score, no matter what sorts of changes take place in its environment. Of course, there are limitations on the nature of the changes. You could, for example, quickly reduce the score to zero by trapping the machine in a small box, but this is really a matter of placing the machine into a situation beyond its ability to cope. That's not fair, even living intelligent creatures have such limitations.

The point is that an Alpha-Class machine adapts rather well to changes in its environment. Its score rarely rises above its chance level, but it is a rather good adaptive mechanism. How do you suppose so many primitive species of living matter have managed to survive millions of years of stress on earth? Functioning on an Alpha-Class level makes them quite adaptive to changes in the environment—they survive. It is a very rudimentary and inefficient sort of survival, but they were here before man appeared, and they might still be here after man disappears.

A Beta-Class machine is more efficient. Adding just a bit of memory does that. The learning curve for your Beta Rodney is different from the simpler Alpha version. Beta Rodney started his existence by making random responses. His initial scoring was at the chance level and, perhaps, even below the chance level. But as the responses are refined, Beta Rodney's score began rising above the chance level. And, if nothing in the environment changes, his ratio of good responses to total encounters rises toward 0.9.

In fact, Beta Rodney can lock himself into a discernable pattern of responses. The pattern will be unique to every lifetime you give him, but you should be able to see Beta Rodney trying to establish a pattern of motion that is peculiar to his developing character and the geometry of the environment.

Rodney might even establish a very simple pattern such as bouncing back and forth between a set of parallel walls. That's all right, he is doing his thing. He is executing a comfortable pattern of behavior that guarantees successful responses. Locking into a particular pattern of motion is, itself, a demonstration of adaptive behavior. Rodney has adapted to the given situation and his learning success is shown by the rising score. Recall that an Alpha-Class machine shows little, if any, change in response scores when its environment is upset in some fashion. Alpha just keeps on plugging away at near-chance levels.

Upset Beta Rodney's pattern, however, and it takes some time for him to recover. The score plunges downward as he encounters the new situation, but that's merely an initial response. With time,

the score begins rising toward 0.9 as he adapts to the new situation and finds a comfortable pattern of responses. Beta Rodney is more upset by changes in the environment than Alpha Rodney is. But Beta Rodney can adapt, and he soon begins outscoring the Alpha version.

The next step in the evolution of this kind of machine intelligence is to come up with a scheme that reduces the drop in scoring as the environment is changed. Such a scheme would be one that can anticipate change and develop some theories for dealing with them. The system would be one that generalizes first-hand experiences and the things it has learned with a high degree of success to situations not yet encountered. So when the environment is changed in some way, the machine does not have to deal with the new features in a random fashion; rather, there will be some hard-won theories at its disposal.

A Gamma-Class machine is one that has this ability to anticipate changes before they occur. As a result of putting this mechanism into your Rodney machine, you will find his learning curve doesn't drop very much when the environment is changed. And what's more, the drop that does occur doesn't last very long. Gamma Rodney establishes a new pattern of motion much more quickly than the Beta version can.

In order to see the Gamma-Class mechanism at work, you have to be prepared to compare the machine's behavior with that of the simpler Beta-Class machine. Let the Gamma Rodney establish a high-scoring pattern of motion, then mess up his little world and see how long it takes for him to adapt.

You will find in this chapter some procedures for bypassing the Gamma programming, leaving only the Alpha and Beta mechanisms. So if you haven't already studied Beta Rodney in the context of adaptive learning curves, you'll be able to do so by eliminating a couple of steps in the program.

In summary, an *Alpha-Class* machine is highly adaptive to changes in its environment. It displays a rather flat and low learning curve, but there is virtually no change in the curve when the environment is altered.

A *Beta-Class* machine demonstrates a rising learning curve that eventually passes the scoring level of the best Alpha-Class machines. If the environment is static, the score eventually rises toward perfection. Change the environment, however, and a Beta-Class machine suffers for a while, the learning curve drops down to the chance level. However, the learning curve gradually rises toward perfection as the Beta-Class machine establishes a new pattern of

behavior. Its adaptive process requires some time and experience to show itself, but the end result is a more efficient machine.

A *Gamma-Class* machine is less upset by changes and recovers faster than the Beta-Class mechanism. This is due to its ability to anticipate changes.

A GAMMA-RODNEY SCHEME

A Gamma-Class scheme is built upon the foundation of Alpha- and Beta-Class behavior. Recall that most Alpha-Class features were incorporated into the Beta Rodney-One program in Chapter 11. Now, we are going to take that Beta Rodney program and add the Gamma Scheme to it.

In fact, the whole Gamma scheme is a subroutine (albeit a very extensive one) that is added to Beta Rodney-One. You will be calling the Gamma subroutine from just a couple of places in the Beta program. Compare the flowcharts in Figs. 11-7 and 12-1.

Since the Gamma routine is a rather extensive and time-consuming operation, it should be called only when it is absolutely needed. In this case, Gamma is called under two particular sets of circumstances: when some Beta-Class response shows a confidence level rising from 2 to 3 and when a confidence level drops from 3 to 2.

Rodney generalizes information only when its confidence level is at the highest level—decimal 3. So whenever some response reaches that level, it is an indication Rodney has some new and high-confidence data to formulate some reasonable theories. Gamma is kicked into the routine the moment a transition to level 3 occurs.

There are going to be many instances, however, where level-3 responses will suddenly fall out of favor, and any suppositions previously based on that information has to be revised accordingly. After all, what good is a set of theories based on information known to be faulty? The Gamma routine is thus called into play whenever any response shows a change in confidence from 3 to 2.

The mechanisms of accomplishing these tasks can be seen in the flowchart in Fig. 12-1. Suppose the system has picked up a response that has a confidence level of 2. The basic information is picked up at BEFORE and later evaluated at CONL=3. If the confidence level is 3, a whole lot of operations are bypassed (including a GAMMA 1 routine) and control goes back to the first RUN ALPHA-PLEX operation. However, we have said that the confidence level in this example is 2, so the output of OK is Y and the subsequent output of the subsequent CONL=3 operation is NO.

The confidence level is next incremented to 3 at INR CONL, and then is updated in the main memory at STORE AS MMDL. Next is another CONL=3 operation. If, at this point, the answer is YES, it must be because the confidence level has just gotten there from a level of 2—if it were a level 3 earlier, we would already be at the top of the chart doing another RUN ALPHA-PLEX.

So a RUN GAMMA 1 takes place after CONL=3 if the response's confidence level has just moved from 2 to 3. In any other sort of increasing-confidence situation, control moves back to top of the chart. This is the mechanism for calling Gamma whenever a response moves from 2 to 3.

The second Gamma operation appears on the NO loop of the OK operation, the OK operation that follows a few steps after BEFORE. In this case, however, the routine is called up only if a confidence level of 2 exists; the CONL=2 can be YES only if it is recently decremented to that level (from level 3) by DCR CONL.

GAMMA PROGRAM FLOWCHARTS

Figure 12-2 is the flowchart for the GAMMA portion of the revised Rodney system. As has been the practice in earlier chapters, we will first define some of the special terms and acronyms before looking at the function of the program in greater detail.

Definitions for the Gamma Subroutine

RUN GAMMA-1—This is the entry point to the entire Gamma scheme. Remember, it is called only when a Beta operation shows either a confidence level reaching its highest point or one dropping from that point.

INIT GAM—"Initialize Gamma" to get things started right. Among other things, this step is responsible for saving all Beta status information in the RAM program stack until the control of all operations returns to Beta once again. Rodney is also told to stop in his tracks at this point.

INIT CYCLE—The Gamma scheme includes a number of loops which can be executed quite a few times. For the sake of this discussion, the major loop is called a *cycle,* and each cycle begins by initializing certain counters and flags which will be described later in this section.

ADDR MMA—"Address the main memory" is the operation here. On the first major Gamma cycle, the main memory is addressed at its highest point, and as a secondary looping operation takes place, the address is decremented until it reaches zero. The system, in other words, looks at every main-memory address.

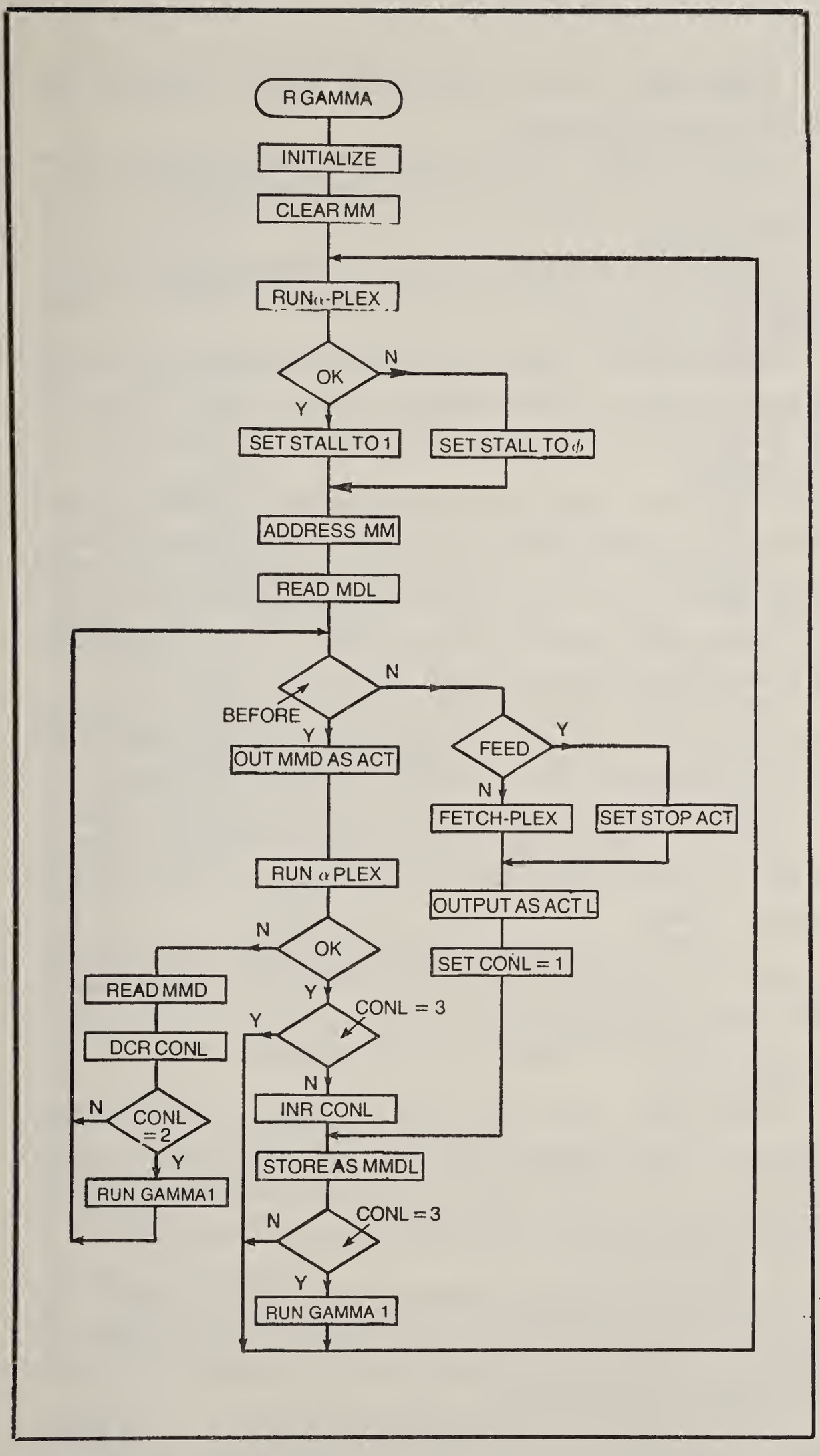


Fig. 12-1. Beta flowchart modified to call Gamma.

MMA REL—"Is the main memory address relevant to the situation being generalized?"

FETCH MMD—If the current main-memory address is indeed relevant, the system then fetches the data contained in that memory location.

CONL=3—"Is the confidence level at that memory location at its highest point?" At this point, the system is searching for high-confidence responses.

MMD REL=0—This step actually evaluates the status of a bit deemed relevant in the high-confidence response code. The answer to the question determines which one of two counters is incremented.

D COUNT—This is a relevant-bit counter that keeps track of the number of logic-0 bits in high-confidence main-memory data.

E COUNT—This is a relevant-bit counter that keeps track of the number of logic-1 bits in high-confidence main-memory data.

END MMA—Has the system checked every main-memory location during the present cycle?

DCR MMA—If the system has not checked every main-memory location during the on-going cycle, the main-memory address is decremented and the check-out cycle is repeated.

D=E—Do the D and E counters contain the same number?

D LT E—Is the D COUNT less than the E COUNT or, in other words, was the number of relevant 1 bits greater than the number of relevant 0 bits?

SET BIT 1—A register is set to insert logic-1 bits into low-confidence data that is, in some way, related to conditions having high-confidence responses.

SET BIT 0—A register is set to insert logic-0 bits into low-confidence data.

INIT MMA—Once the system determines the values of bits relevant to high-confidence responses, the main memory is again recycled to insert these bits into low-confidence locations in the main-memory data. Here, the main-memory address is initialized to begin this sequence.

CONL LT 2—Is the confidence level of the main-memory data less than 2; in other words, is it 1 or 0? This step is searching for data having low confidence levels, presumably to upgrade them with high-confidence information.

SET CONL = 1—Set the confidence level in 1. All stored responses that are built up by the generalizing procedure carry an initial confidence level of 1.

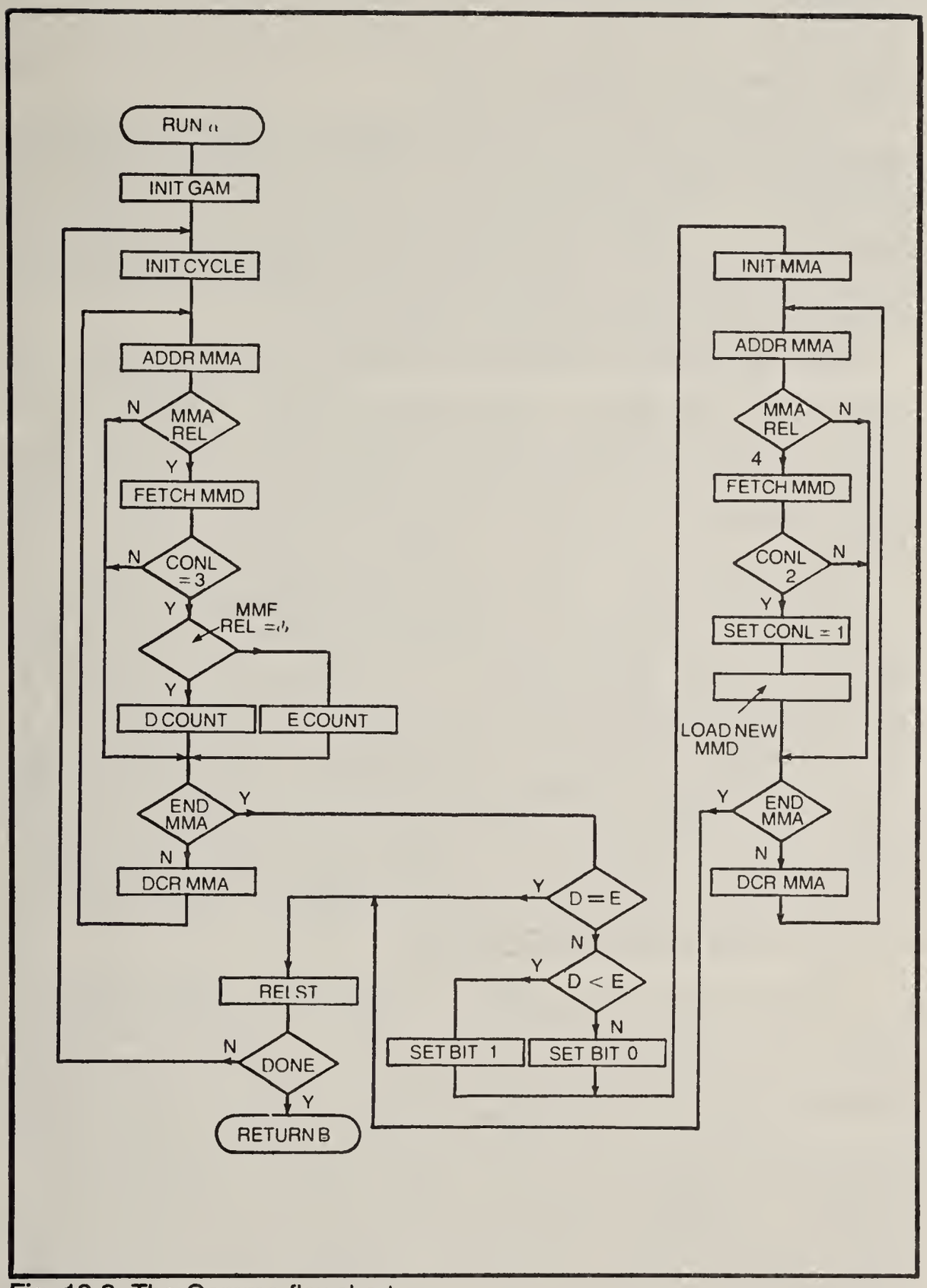


Fig. 12-2. The Gamma flowchart.

LOAD NEW MMD—Store the new data and confidence level in the main memory.

RELST—This is the subroutine that dictates the nature of the confidence and relevance searches. It is the weakest link in the system in the sense that it carries some of the experimenter's prejudices and limitations. If the experimenter isn't careful in the design of this particular subroutine, the machine's entire character will be molded along the lines of programmed parabots.

DONE—Is the Gamma subroutine done?

RETURN BETA—Machine control is returned to the Beta program.

General View of the Gamma Program

The system enters the Gamma sequence whenever there is a change in any high confidence level status. It makes no difference at all what sort of response and environmental condition caused the change—the important thing is that any change in any confidence level from 2 to 3 or from 3 down to 2 implies a need for altering some theories about conditions not yet experienced.

The particular Gamma program included in this chapter first searches the main theory for any address having a 0 in the MMA0 (lowest-order) bit position and carrying response data having a confidence level of 3. Of course, there are going to be a whole lot of MMAs ending in 0, half of them, in fact, but there normally won't be many ending in 0 and carrying data with a confidence level of 3.

Now, if you recall that each main-memory address corresponds to a particular environmental condition, you should be able to see that searching for MMAs ending in 0 is tantamount to looking for environmental conditions where the LMR (left motor reverse) bit of ENVL is at logic 0. So what we are doing on this first pass through the Gamma program is looking for environmental conditions where LMR was at 0 when something happened to prompt a response. If that condition had occurred often enough, and if it has been successfully dealt with at least three times in succession, the response is worthy of being considered "relevant."

The Gamma program scans the main memory, systematically searching for addresses and data that meet these "relevance criteria." For the sake of this discussion, suppose it finds some information that fits. Upon finding an MMA and response data which meet the demands of the relevance criteria, the system looks at the relevant bit position of the stored, high-confidence response. In this example, the system is looking at the lowest-order MMA bit; so when it finds a point of relevance, it looks at the lowest-order response bit in the main-memory data. If this particular data bit happens to be a logic 1 (indicating the condition was met with a response code (ACTL) having a logic 1 in the LMR (D0) position, a 1s counter is incremented. If, on the other hand, the successful response had a LMR bit of 0, a 0s counter is incremented.

The system looks for environmental conditions, of a particular type, having high-confidence responses and then keeps track of the number of 1s or 0s in the response codes. After this first phase of the operation is completed, the two counters are compared to see which

logic level occurred more often. If there were no responses meeting the relevance criteria, or if there were an equal number of 1s and 0s responses, the results are discarded and another set of relevance criteria is established.

Assume, however, that there were some valid, high-confidence responses and that there were more 1s than 0s meeting the criteria. This being the case, the system then scans the main-memory address looking for low-confidence levels (1 or 0) that have the same environmental criteria. When such a condition is found, a 1 bit is stored into the low-order response position and the confidence level is set to 1. There is generally a rather large number of such places in the main memory.

So whenever most environmental conditions (having the lowest-order ENVL bit equal to 0) are successfully solved, most of the time, with a 1 bit in the lowest-order ACTL response, this 1 bit is stored in ACTL response positions for ENVLs not successfully solved when the lowest-order ENVL bit is 0. After loading these speculations into low-confidence response positions, the criteria for relevance is changed to the ENVL condition where the lowest-order bit is 1 instead of 0, and the whole process starts all over again. After that, the relevance bit is changed from the first to the second bit of MMA. It is run again, looking for 0s and then again looking for 1s. This process continues until most of the bit positions have been checked for relevance, the results of high-confidence bits compiled and stored in low-confidence positions, and Rodney's conjectures, concerning ways to respond to unfamiliar situations, are as complete as his high-confidence knowledge allows.

Indeed, this is something of an abstract process. It does not cause any immediate responses anyone can detect. Rather, it is something akin to thought processes that cannot be detected or evaluated until they take some overt form at a later time.

It probably seems sort of weird. We have no real control over it and we cannot see, in a direct way, exactly what is going on. That's the nature of this self-programming process. We could attach a printer or CRT to the memory and display the Gamma activity, but we would be either buried in a mountain of paper or dazzled with a blur of characters on the screen. If you like, you can attach a radio receiver to Rodney and pick up the r-f signals from the Gamma program. The sound—the clicking and squealing—reminds one of EEG activity monitored at the skull of living creatures.

The only way to get a handle on the operation of the Gamma program is by testing Rodney's adaptive behavior. Even then, there is no way to tell exactly how he arrived at the conclusions he did, and

that's what makes this scheme different from most on-going work in artificial intelligence. There is no such thing here as a programming "tree." The environment and Rodney's initial responses to it set the pace, but each encounter with the environment further modifies and refines all activity.

It must be pointed out that this is only a rudimentary relevance-checking procedure. It is workable, but it might not be the best possible one for this system. If I had chosen to research Gamma relevance selection thoroughly before preparing this book manuscript, you probably wouldn't be reading it for another year or so. Gamma relevance-selection theory is a subject in its own right that might fill a book one day. I certainly intend to do more research along this line, and I would certainly encourage other experimenters to do the same.

RELST Flowchart

The keystone element in the relevance-selection operation is the RELST (relevance set) subroutine. This subroutine is blocked out in Fig. 12-3. While this is a workable scheme, I believe it could use more work. Any takers?

The main Gamma program is initialized to look for a 0 bit in MMA. This bit is carried into RELST as PHBYT (PHase BYTe), and since this flag byte is set to 0 initially, it follows that the answer to PHBYT=1 will be N the first time around.

With an N output from PHBYT, the system goes to SET PHBYT=1, where the phase byte is changed to 1, a not-zero flag is set at SET NZ (indicating the Gamma operation isn't complete), and then the program returns to the main Gamma program.

The Gamma program then cycles through the same sequence as before, looking for a 1 bit in MMA instead of a 0. When RELST is called this time, the answer to PHBYT=1 is Y and it is time to shift the relevance bit position. The first RELST operation changed the value of the relevance bit from 0 to 1, but now the position of the relevance bit is changed and the value is returned to 0.

Since this program is being used in conjunction with a Beta format, which works with only the 8 lower bits of MMA, the MMA bits tested for relevance include only the lower 8 bits or byte. Even then, there is nothing gained by checking relevance for all eight of them. Here, we are going to check the lower 7 which, translated into terms of ENVL, means all environmental conditions except $\overline{\text{HUNGRY}}$.

The 4 lower-order bits (or nibble) are checked for relevance one at a time, and never more than one at a time. In other words, we

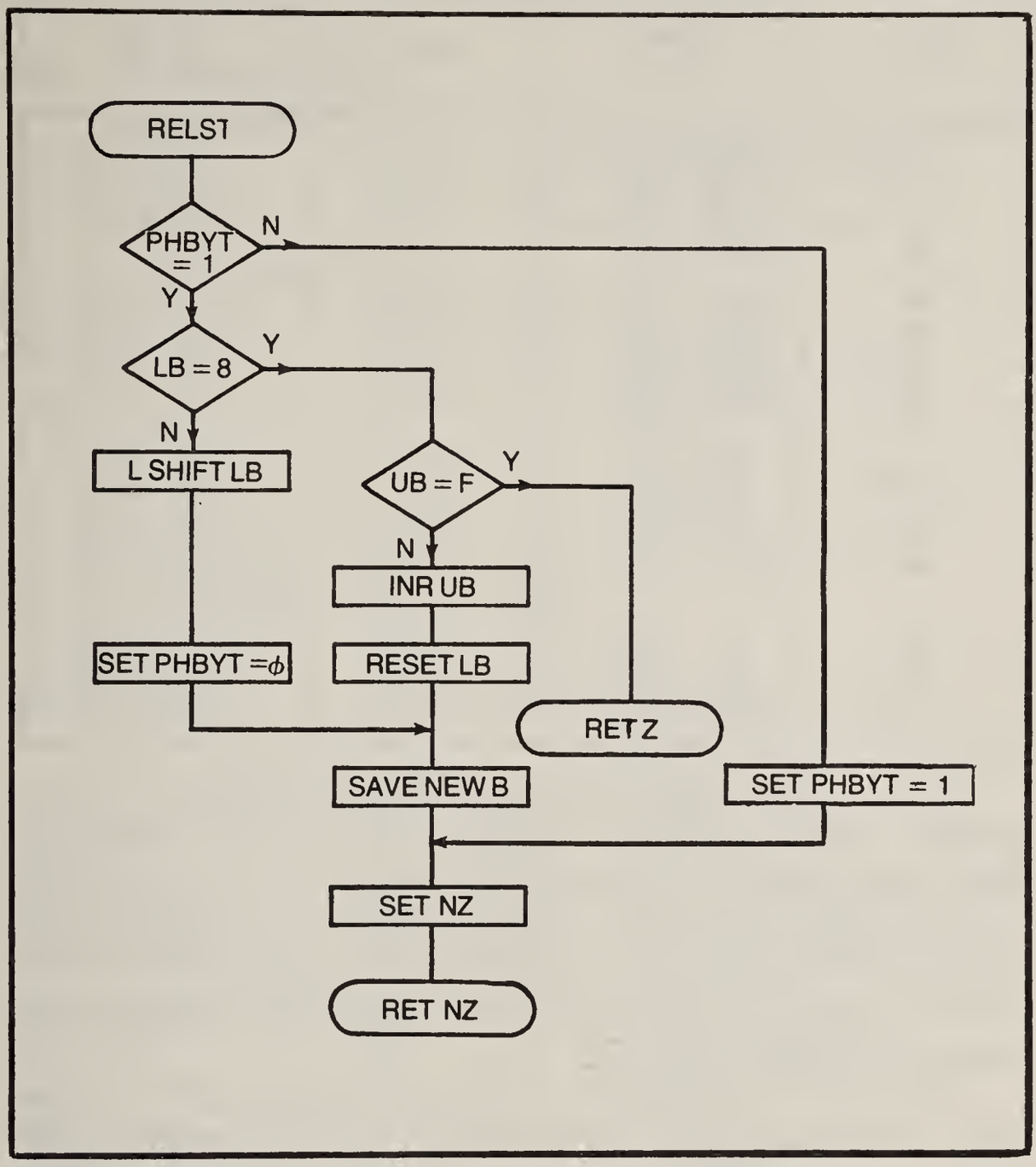


Fig. 12-3. Flowchart for the RELST Gamma subroutine.

are going to check LMR, LMF, RMR and RMF of ENVL separately. The hex count is: 0, 1, 4, and 8.

The 3 higher-order bits used in this scheme will be checked systematically and often more than one at a time. The high-order HUNGRY bit will always be fixed at logic 1, so the upper nibble relevance sequence follows the hex counting code: 8, 9, A, B, C, D, E, F.

Putting together the upper and lower nibbles of the relevance byte, the relevance check proceeds as shown in Table 12-1, where a logic-1 level signifies a bit being checked for relevance.

The system thus cycles through 32 different relevance bytes, but remember that each LB is first tested for 0s and then for 1s, effectively doubling the number of cycles to 64. This can consume a lot of time, especially when the machine has to deal with a lot of high-confidence situations.

Table 12-1. Summary of relevance-select operations.

Hex	Binary UB	Binary LB	Hex	Binary UB	Binary LB
81	1000	0001	C1	1100	0001
82	1000	0010	C2	1100	0010
84	1000	0100	C4	1100	0100
88	1000	1000	C8	1100	1000
91	1001	0001	D1	1101	0001
92	1001	0010	D2	1101	0010
94	1001	0100	D4	1101	0100
98	1001	1000	D8	1101	1000
A1	1010	0001	E1	1110	0001
A2	1010	0010	E2	1110	0010
A4	1010	0100	E4	1110	0100
A8	1010	1000	E8	1110	1000
B1	1011	0001	F1	1111	0001
B2	1011	0010	F2	1111	0010
B4	1011	0100	F4	1111	0100
B8	1011	1000	F8	1111	1000

Because of this time delay required for generalizing good responses, I decided to insert a stop code at ACTL during the Gamma-running interval. This isn't absolutely necessary, one might have the system call up Beta at the end of each cycle to see if a stall condition exists; if it does, the current Gamma status could be saved in the memory stack until the stall condition is cleared. Gamma could then pick up where it left off.

I certainly encourage a knowledgeable experimenter to try this idea. It works much like a microprocessor interrupt sequence. We cannot use an interrupt here, however, because the stall conditions are sensed by a software routine.

THE GAMMA PROGRAMS

The Gamma Rodney-One system can be considered an extension of the Beta version. Assuming you have already loaded the Beta routines from Chapter 11 and have them working, you need only add a few steps to the main Beta program and load the Gamma subroutine to get this new and exciting system going.

The first program listed here is called *Beta With Gamma Calling*. Most of the steps are already loaded into the system as part of the main Beta program, but note the extra ones that replace the NOPs (no-operations) which appeared in the main Beta program. Just add the extra steps at address locations 043H through 049H 052H, and 058H to complete this part of the job.

The important Gamma subroutine is listed here as *The Main Gamma-1 Program.* This is all new programming.

Beta With Gamma Calling

```
                        BETCG:
000 31  FF 03           LXI     SP, TOPS ;SET STACK POINTER
003 CD  00 01           CALL    CLRM     ;"CLEAR MM"
006 21  00 08           LXI     H,PORTO  ;SET PORT POINTER
                                           TO PORT 0
009 CD  50 01 ;FETEN    CALL    RUAPX
00C 7E                  MOV     A, M     ;FETCH ENVL
00D C2  15 00           JNZ     SSB1     ;IF "OK" IS Y, JUMP TO
                                         ;"SET STALL TO 1"
010 E6  CF              ANI     CFH      ;ELSE SET STALL BITS
                                           TO 0
012 C3  17 00           JMP     AMM      ;AND JUMP TO "ADDRESS
                                           MM"
015 F6  30      ;SSB1   ORI     30H      ;SET STALL BITS TO 1
017 32  04 08 ;AMM      STA     PORT4    ;
01A 3A  06 08 ;SAVD     LDA     PORT6    ;"READ MDL"
01D 47                  MOV     B,A
01E E6  C0              ANI     C0H      ;ISOLATE CONL BITS
020 CA  5C 00           JZ      FEDUP    ;IF "BEFORE" IS N,
                                           JUMP TO
                                         ;"FEED"
023 78                  MOV     A, B     ;ELSE FETCH MMD FROM B
024 F6  F0              ORI     FOH      ;CONDITION ACTL
026 77                  MOV     M,A      ;AND "OUT MMD AS ACT"
027 CD  50 01           CALL    RUAPX    ;CALL "RUN ALPHA-PLEX"
02A CA  TP 00           JZ      RMMD     ;IF "OK" IS N, JUMP TO
                                         ;"READ MMD"
02D 3A  06 08           LDA     PORT6    ;ELSE FETCH MMD
030 E6  C0              ANI     C0H      ;ISOLATE CONL BITS
032 FE  C0              CPI     COH      ;"CONL=3"—Z IF YES,
                                         ;NZ IF NO
034 CA  09 00           JZ      FETEN    ;LOOP BACK TO START
                                           OF CYCLE
                                         ;IF YES
037 C6  40              AID     40H      ;ELSE "INR CONL"
039 4F                  MOV     C, A
03A 3A  06 08           LDA     PORT6    ;FETCH MMD
03D E6  3F              ANI     3FH      ;SET CONL BITS TO ZERO
03F B1                  ORA     C        ;COMPOSE NEW MMD WORD
040 32  06 08 ;SAMMD    STA     PORT6    ;"STORE AS MMDL"
043 E6  C0              ANI     C0H      ;ISOLATE CONL BITS
045 FE  C0              CPI     C0H      ;"CONL=3"—Z IF YES,
                                         ;NZ IF NO
047 CC  00 02           CZ      GAMMA1   ;IF YES, CALL GAMMA-1
                                         ;SUBROUTINE
04A C3  09 00           JMP     FETEN    ;LOOP BACK TO START
                                           OF CYCLE
04D 3A  06 08 ;RMMD     LDA     PORT6    ;FETCH MMD
050 D6  40              SUI     40H      ;"DCR CONL"
052 E6  C0              ANI     C0H      ;ISOLATE CONL BITS
054 FE  40              CPI     40H      ;"CONL=2"—Z IF YES,
                                         ;NZ IF NO
```

```
056 CC 00 02          CZ      GAMMA1  ;IF YES, CALL GAMMA-1
                                      ;SUBROUTINE
059 C3 1A 00          JMP     SAVD    ;RETURN TO "BEFORE"
05C 3A 04 08 ;FEDUP   LDA     PORT4   :FETCH MMA/ENVL
05F E6 80             ANI     80H     ;ISOLATE FEED BIT
                                      ;Z IF YES, NZ IF NO
061 C2 68 00          JNZ     FETPL   ;IF N, JUMP TO
                                        "FETCH-PLEX"
064 AF                XRA     A       ;CLEAR ACCUMULATOR
065 C3 6B 00          JMP     OUACT   ;JUMP TO "OUTPUT AS
                                        ACTL"
068 CD 20 10 ;FETPL   CALL    FETR
06B 77       ;OUACT   MOV     M,A     ;"OUTPUT AS ACTL"
06C F6 80             ORI     80H     ;"SET CONL=1"
06E C3 40 00          JMP     SAMMD   ;JUMP TO "STORE AS
                                        MMDL"
                                      ;END OF BETCG PROGRAM
              INIT:
071 3E F0             MVI     A,F0H   ;SET STOP CODE
073 32 00 08          STA     PORT0   ;LOAD STOP CODE AS
                                        ACTL
076 21 04 08          LXI     H,PORT4 ;SET PORT POINTER TO
                                        MMAL
079 C9                RET             ;RETURN
```

The Main Gamma-1 Program

```
              GAMMA1:
300 F5                PUSH    PSW     ;SAVE BETA STATUS
301 CD 71 00          CALL    INIT    ;CALL "INITIALIZE"
304 06 81             MVI     B,FREL  ;INITIALIZE RELEVANCE
                                        BYTE
306 AF       ;STAR    XRA     A
307 32 F0 02          STA     PHBYT   ;CLEAR PHASE BYTE
                                        TO ZERO
30A 57                MOV     D,A     ;CLEAR D COUNTER
30B 5F                MOV     E,A     ;CLEAR E COUNTER
30C 0E                MVI     C,FFH   ;INITIALIZE MMA POINTER
30D CD       ;MMACK   CALL    MARL
310 CA 16 03          JZ      FMMD    ;IF Y, JUMP TO
                                        "FETCH MMD"
313 C3 32 03          JMP     EDMA    ;ELSE JUMP TO "END MMA"
316 3A 06 08 ;FMMD    LDA     PORT6
319 E6 C0             ANI     C0H     ;ISOLATE CONL BITS
31B FE C0             CPI     C0H     ;"CONL=3"—IF Y, Z
                                      ;IF N, NZ
31D C2 32 03          JNZ     EDMA    ;IF N, JUMP TO
                                        "END MMA"
320 78                MOV     A,B     ;ELSE FETCH RELEVANCE
                                        BYTE
321 E6 0F             ANI     0FH     ;ISOLATE LOW-ORDER
                                        NIBBLE
323 21 06 08          LXI     H,PORT6 ;SET PORT POINTER
                                        TO MMD
326 A6                ANA     M       ;"MMD REL=0"—IF Y, Z
                                      ;IF N, NZ
```

```
327 21 06 08          LXI   H,PORT4 :RETURN PORT POINTER TO
                                    ;MMA
32A CA 31 03          JZ    DCON    ;IF Y, JUMP TO
                                        "INR D COUNT"
32D 1C                INR   E       ;ELSE "INR E COUNT"
32E C3 32 03          JMP   EDMA    ;JUMP TO "END MMA"
331 14          ;DCON INR   D       ;"INR D COUNT"
332 AF          ;EDMA XRA   A
333 B9                CMP   C       ;"END MMA"—IF Y, Z
                                    ;IF N, NZ
334 CA 3B 03          JZ    DEQE    ;IF Y, JUMP TO "D=E"
337 0D                DCR   C       ;ELSE "DCR MMA"
338 C3 0D 03          JMP   MMACK   ;AND JUMP BACK TO
                                        "ADDR MMA"
33B 7A          ;DEQE MOV   A,D     ;FETCH D COUNT
33C BB                CMP   E       ;COMPARE WITH E COUNT,
                                        "D=E"
                                    ;IF Y, Z
                                    ;IF N, NZ
33D C2 48 03          JNZ   DLTE    ;IF N, JUMP TO "D LT E"
340 CD          ;RLST CALL  RELST   ;ELSE CALL "REL RESET"
343 C2 06 03          JNZ   STAR    ;IF NOT DONE, RESTART
346 F1                POP   PSW     ;ELSE RECOVER
                                        BETA STATUS
347 C9                RET           ;RETURN CONTROL TO BETA
348 DA 50 03 ;DLTE    JC    SB1     ;IF Y, JUMP TO "SET BIT 1"
34B 16 00             MVI   D,00H   ;SET LOWER NIBBLE
                                        OF D TO 0'S
34D C3                JMP   LOAD    ;AND JUMP TO LOAD ROUTINE
350 16 0F       ;SB1  MVI   D,0FH   ;SET LOWER NIBBLE
                                        OF D TO 1'S
352 0E FF       ;LOAD MVI   C,FFH
354 CD          ;MACK2 CALL MARL    ;CALL "MMA REL"
357 CA 5D 03          JZ    FMMD2   ;IF Y, JUMP TO "CONL LT 2"
35A C3                JMP   EDMA2   ;ELSE JMP TO "END MMA"
35D 3A 06 08 ;FMMD2   LDA   PORT6
360 E6 C0             ANI   C0H     ;ISOLATE CONL BITS
362 FE 80             CPI   80H     ;"CONL LT 2"—IF Y, C
                                    ;IF N, NC
364 D2                JNC   EDMA2   ;IF N, JUMP TO "END MMA"
367 78                MOV   A,B     ;ELSE FETCH RELEVANCE
                                        BYTE
368 2F                CMA           ;COMPLEMENT
369 F6 F0             ORI   F0H     ;SET UPPER NIBBLE TO 1'S
36B 5F                MOV   E,A     ;SAVE RESULT IN E
36C 3A 06 08          LDA   PORT6   ;FETCH MMD
36F A3                ANA   E       ;CLEAR THE BITS TO BE
                                        CHANGED
370 32 06 08          STA   PORT6   ;RETURN TO MMD
373 7A                MOV   A,D     ;FETCH BIT STATUS
374 A0                ANA   B       ;ISOLATE RELEVANT BIT
375 E6 0F             ANI   OFH     ;ISOLATE LOWER NIBBLE
377 F6 40             ORI   40H     ;"SET CONL=1"
379 5F                MOV   E,A     ;SAVE IN E
37A 3A 06 08          LDA   PORT6   ;FETCH MMD
37D B3                ORA   E       ;CHANGE RELEVANT AND
                                        CONL BITS
```

```
37E  32   06 08          STA     PORT6   ;"LOAD BITS"
381  AF          ;EDMA2   XRA     A
382  B9                   CMP     C       ;END MMA—IF Y, Z
                                          ;IF N, NZ
383  CA                   JZ      RLST    ;IF Y, JUMP TO
                                              RESTART CYCLE
386  0D                   DCR     C       ;ELSE "DCR MMA"
387  C3   54 03           JMP     MACK2   ;AND JUMP BACK TO
                                              "ADDR MMA"
38A  C7                   RST     0       ;RESET TO ZERO
                                          ;END OF GAMA1
```

REVERTING BACK TO BETA

If, for experimental reasons, you ever want to eliminate the Gamma subroutine after entering it, do the following:

```
LOAD   043   C3   4A   00
       052   C3   59   00
```

These two unconditional JUMP operations bypass the Gamma portion of the program. Of course, getting back the Gamma operation is a matter of replacing the data:

```
LOAD     043   E6   C0
         052   E6   C0
```

Adding A Cassette Tape Interface

All the time you have spent entering programs into the program RAM can be lost in a fraction of a second if the power supply fails for any reason. Of course, you have a nice backup system if you keep Rodney at the battery charger and running from the auxiliary power supply most of the time, but just knowing that all the programming can be lost is a scary prospect.

One way to back up the backup system is by keeping the programs, the information in the program RAM, stored on magnetic tape. If there is any loss of program information, it can be reloaded from the tape in just a couple of minutes.

The cassette tape interface described here was originally devised by Intel to demonstrate the value of the serial inputs and outputs that are uniquely available on the 8085 microprocessor chip. The procedure assumes the tape machine is of a rather low quality and uses a tone-burst/no-tone ratio to encode 1s and 0s. Thus, the tape can stretch, the machine can run at varying speeds, and yet the data can be retrieved reliably.

The interface circuit is also very simple and inexpensive. It uses just one IC and a mere handful of ordinary resistors and capacitors. But here's the catch—The scheme is so simple because it uses microprocessor software, or programming, to generate the tones, decode the tones, and keep the data flowing back and forth between the tape player and microprocessor in an orderly fashion. However, this type of system means you will have to burn a PROM to store the tape program. (Otherwise that program could get scrambled, too).

The PROM suggested here is a 1702, 256X8 ROM. I selected this one because it is inexpensive and widely available, not because it happens to be the easiest to use. Most of the major electronics magazines though have published articles showing how to build a ROM programmer for the 1702.

CASSETTE TAPE INTERFACE CIRCUIT

The circuit for the cassette interface is shown in Fig. 13-1. This circuit can be added in some unused space on the I/O board. You can set up the pin numbers in any convenient way, but this diagram shows SOD going to pin L on the I/O board, then to the SOD line on the CPU board via pin L on the CPU board. SID is likewise connected to pin B on both boards and to the SID pin on the microprocessor chip.

Pin K on the I/O board goes to the "hot" lead of a plug leading to the AUX input on the cassette tape machine. In a similar way, pin C goes to the "hot" lead from the earphone jack on the tape machine. Don't forget to connect the COMM leads of these two plugs to Rodney's COMM.

There is nothing at all tricky about this circuit. It is a set of ordinary audio amplifiers which condition the signal levels and match impedances between the digital system and the cassette tape machine. The microprocessor generates the burst tones, compiles a leader, and all that sort of thing. A parts list for the tape interface circuit is shown in Table 13-1.

ADDING THE PROM TO THE CPU BOARD

We haven't dealt with the tape program yet, but after you've had a chance to try it out for yourself, you can "burn" it into the 1702 PROM. In the meantime, you can install the socket for this PROM on the CPU board. Figure 13-2 shows the wiring diagram, and Fig. 13-3 shows where the PROM should be situated on the CPU board.

Table 13-1. Parts list for the tape interface circuit and tape program ROM.

Z210 LM324 quad operational amplifier (Jameco)
1 ea. 14-pin WW socket (Jameco)
1 ea. 100 Ohm resistor
1 ea. 220 Ohm resistor
1 ea. 820 Ohm resistor
3 ea. 1k resistor
6 ea. 10k resistor
1 ea. 22k resistor
1 ea. 1 Meg resistor
1 ea. 0.001 μF mylar capacitor
3 ea. 0.1 μF mylar capacitor
2 ea. 1N458 rectifier diode (Jameco)

PROM Components to the Added to CPU Board (Board 100)

Z131 1702 256×8 PROM (Jameco)
1 ea. 24-pin WW socket (Jameco)

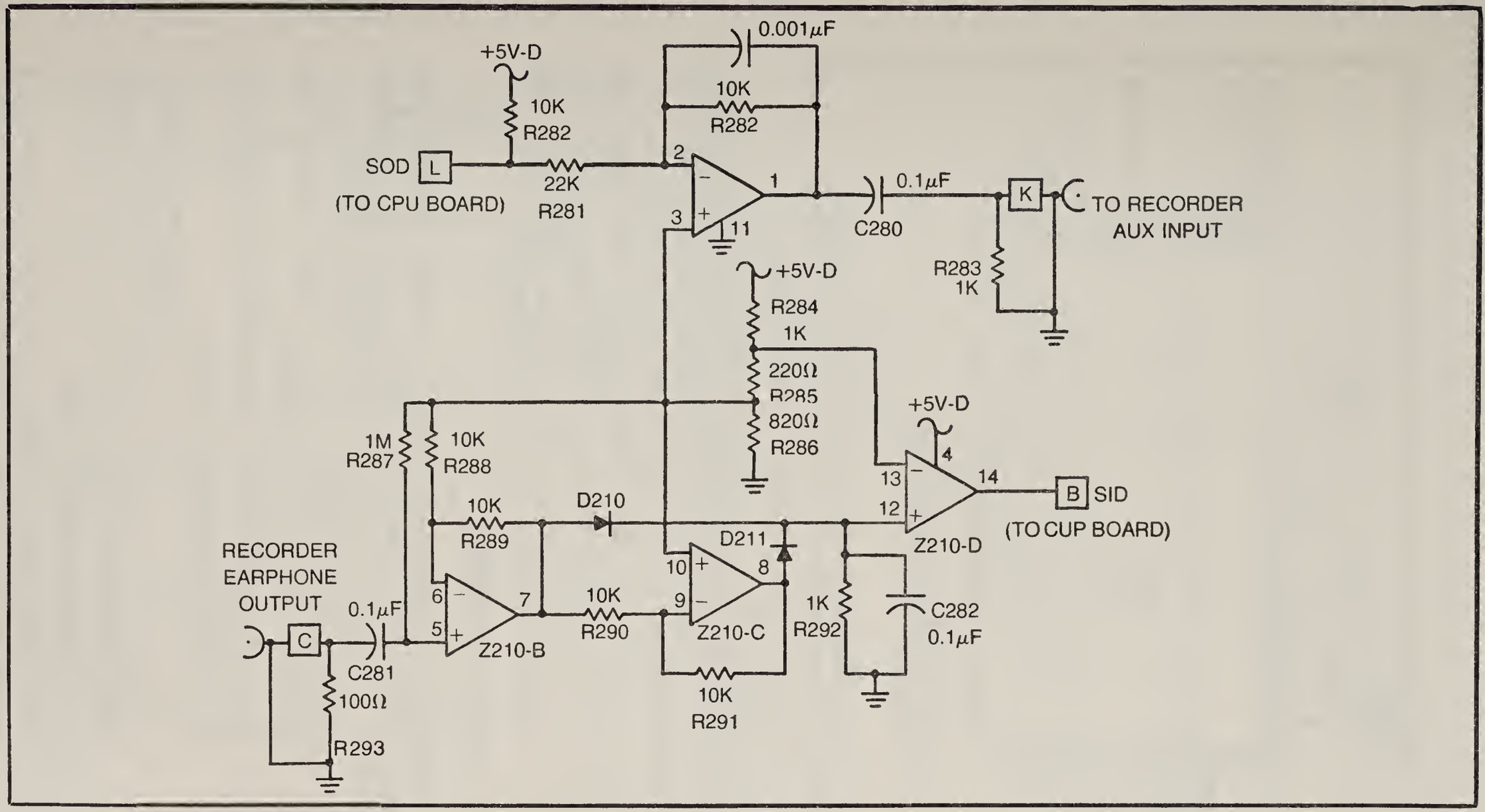


Fig. 13-1. The tape interface circuit.

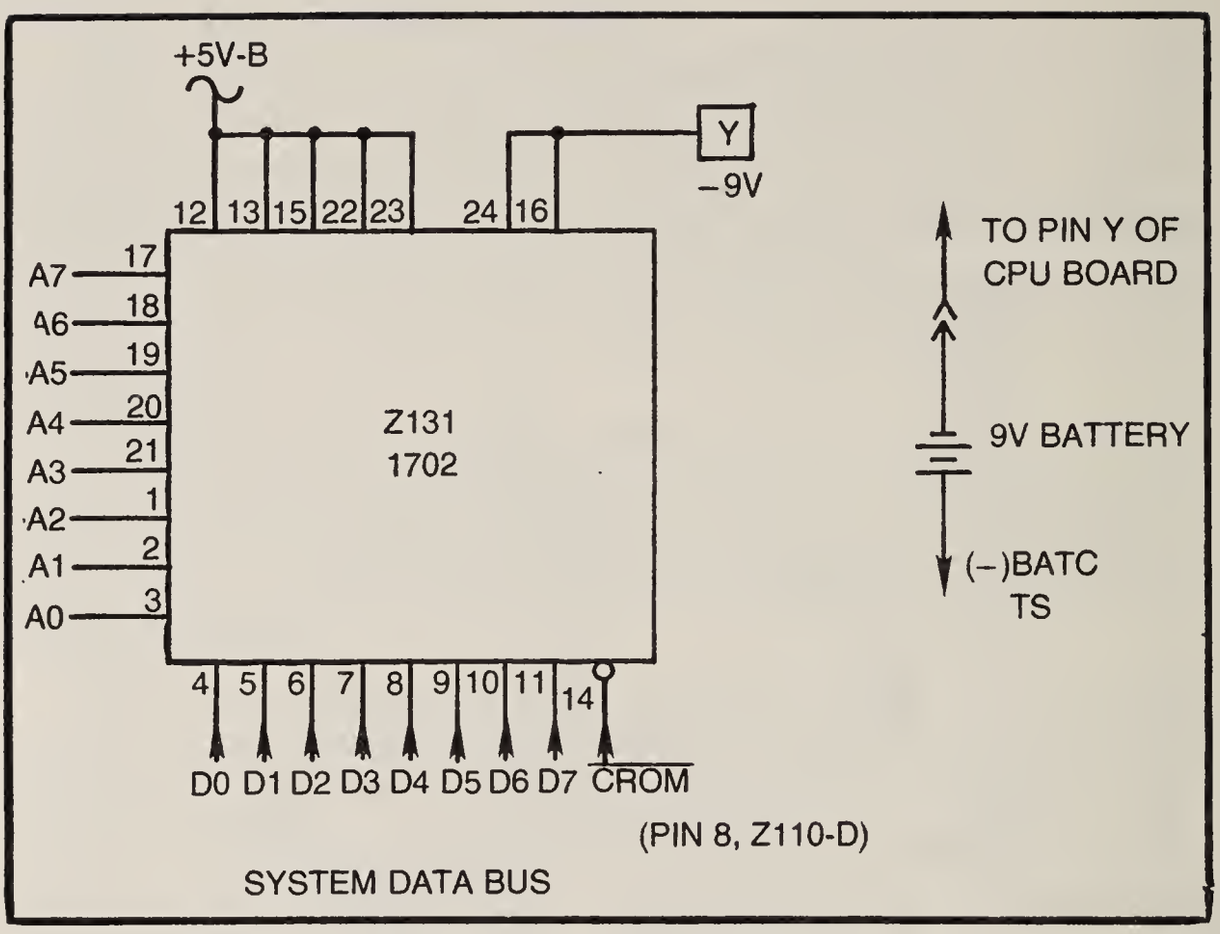


Fig. 13-2. The tape program ROM circuit on the CPU board.

I have already mentioned that the 1702 isn't the easiest ROM to use, and this fact shows up in the little battery circuit in Fig. 13-2. The PROM requires a −9V supply for normal operation.

You are free, of course, to devise a little power supply that supplies −9V, but a 9V battery can be used just about as easily. The battery doesn't have to be connected in the circuit at all times—just plug it in whenever you plan to use the tape interface.

USING THE TAPE INTERFACE

Suppose you have some program information stored in the program RAM, and you want to save it on cassette tape. Assuming the ROM is "burned" and installed, here is the procedure:

1. Connect the tape machine to the interface circuit and connect the −9V supply to the PROM.
2. To get the microprocessor running from the PROM you must LOAD the following program:

   ```
   000  C3  00  04      JMP PROM; JUMP TO THE PROM ADDRESS
   ```

   Of course this "writes over" the first three instructions in your stored RAV program, but you can replace them after the recording operation is done.
3. Turn on the tape recorder (in the RECORD mode, of course), set the system to RUN and turn off the RESET switch.

4. About 40 seconds later, the motor status lamps on the power-distribution panel will begin flashing to indicate the entire contents of the program RAV is on the tape. Turn off the tape recorder immediately.
5. Return the system to PROGRAM and turn on the RESET switch
6. Restore the first three instructions in the program (the ones you "wrote over" in step 2), and you're back in business.

Here's how you can program the system RAM from the tape:

1. Connect the tape machine to the interface circuit and connect the −9V supply to the PROM.
2. LOAD the following program in the program RAM

```
000  C3  A0  04      JMP PRMPL; JUMP TO THE PLAYBACK
                                ;INSTRUCTION IN THE PROM
```

   Again, this step destroys the first three instructions in your program RAM. But, as before, you can restore them yourself.
3. Turn on the tape machine (in the PLAYBACK mode), set the system to RUN, and turn off the RESET switch.

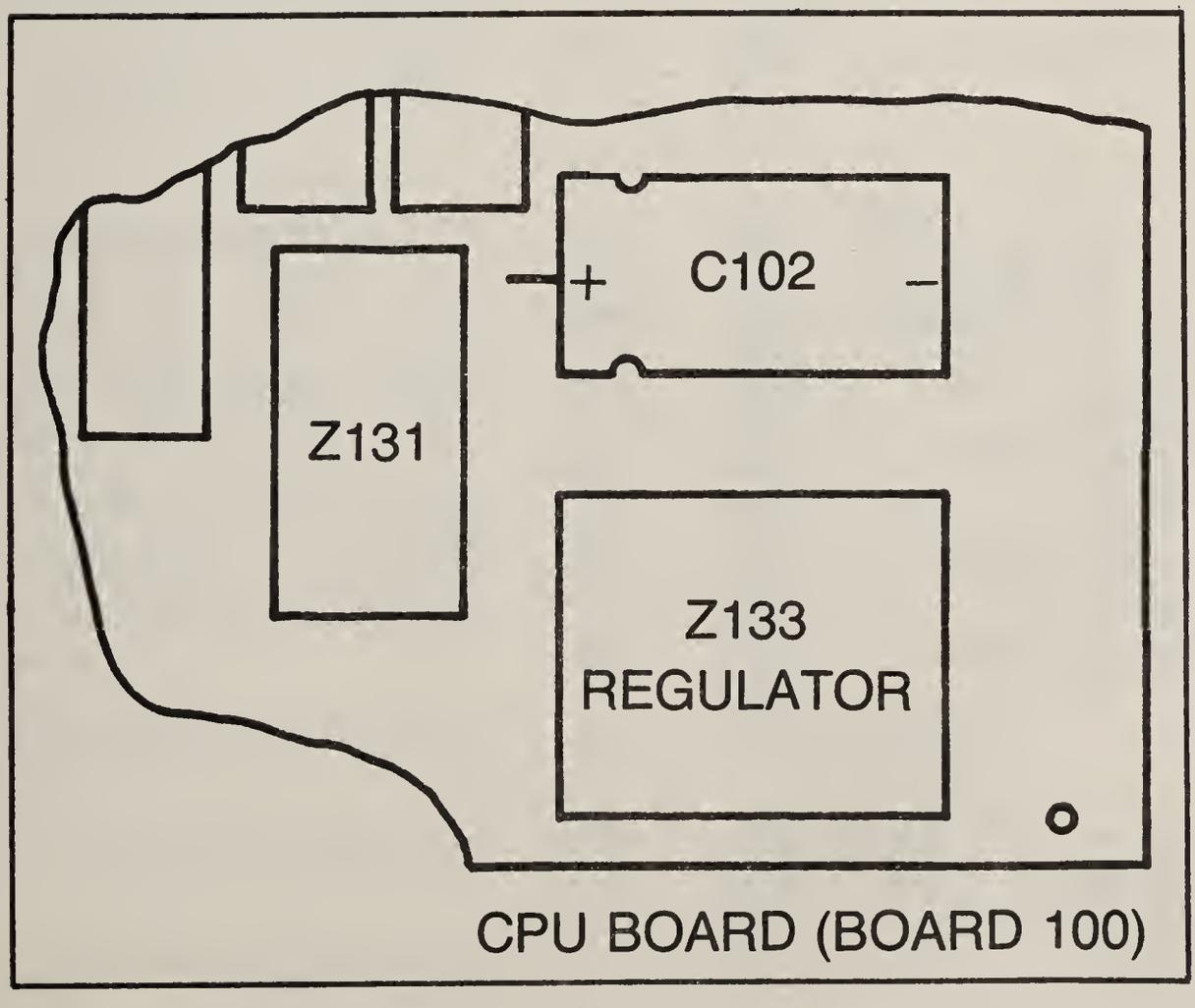


Fig. 13-3. Layout of the program ROM on the CPU board.

4. When the playback job is done, the motor status lamps will flash. Turn off the tape machine right away.
5. Return the system to PROGRAM and turn on the RESET switch.
6. Program the first three instructions in your program format, and the system is ready to go again.

Both the recording and playback operations should be carried out with the level control on the tape player set for maximum volume. In case you're interested, the burst tone has a frequency of about 3kHz and the bit-transfer rate is about 100 bps.

THE TAPE-INTERFACE PROGRAM

The following program is preset using ROM addresses. It can be modified to operate in the RAM space, but if you want to try that trick, remember to change all CALL and JUMP instructions to the appropriate addresses.

Cassette Record—ROM Version

```
400  31  03 FF  RECO:   LXI    SP, TOPS  ;SET STACK POINTER TO
                                          ;TOP OF RAM SPACE
403  21  00 00           LXI    H,0000H   ;INITIALIZE MEMORY
                                          ;POINTER
406  CD  10 04  CBKRD:   CALL   BKRCD     ;CALL RECORD SEQUENCE
409  23          INR    H                 ;INCREMENT H COUNTER
40A  FE  03         CPI    HMMAX         ;
40C  C2  06 04      JNZ    CBKRD         ;IF NOT DONE, JUMP BACK
                                          ;TO RECORD SEQUENCE
40F  CD  70 04      CALL   FLASH         ;ELSE CALL FLASH TO
                                          ;SIGNAL END OF RECORDING
                    END                  ;


410  0E  FA     BKRCD:  MVI    C,FAH     ;SET LEADER BURST LENGTH
412  3E  C0             MVI    A,C0H
414  CD  30 04  BR1:    CALL   BURST     ;OUTPUT TONE
417  0D              DCR    C
418  C2  14 04       JNZ    BR1           ;SUSTAIN LEADER TONE
41B  AF             XRA    A             ;CLEAR ACCUMULATOR
41C  CD  30 04      CALL   BURST
41F  4E         BR2:    MOV    C,M       ;FETCH DATA BYTE TO BE
                                          ;RECORDED
420  CD  40 04      CALL   CASO          ;OUTPUT BYTE TO RECORDER
423  2C              INR    L             ;POINT TO NEXT BYTE
424  C2  1F 04       JNZ    BR2
427  C9              RET
                                          ;END OF BKRCD SUBROUTINE


430  16  10     BURST:  MVI    D,10H     ;SET NUMBER OF CYCLES
432  30         BU1:    SIM
433  1E  1E             MVI    E,1EH
435  1D         BU2:    DCR    E         ;REGULATE TONE FREQUENCY
```

```
436 C2 35 04        JNZ   BU2
439 EE 80           XRI   80H      ;COMPLEMENT SOD DATA BIT
43B 15              DCR   D
43C C2 32 04        JNZ   BU1      ;CONTINUE UNTIL BURST OR
                                   ;PAUSE IS FINISHED
43F C9              RET            ;RETURN
                                   ;END OF BURST SUBROUTINE


440 F3        CASO: DI             ;DISABLE INTERRUPTS
441 D5              PUSH  D        ;SAVE D IN STACK
442 06 09           MVI   B,09H    ;SET BYTE SIZE
444 AF        C01:  XRA   A        ;CLEAR ACCUMULATOR
445 3E C0           MVI   A,C0H    ;SET A FOR TONE BURST
447 CD 30 04        CALL  BURST    ;
44A 79              MOV   A,C      ;FETCH NEXT BIT TO
                                       BE RECORDED
44B 1F              RAR            ;ROTATE TO SOD POSITION
44C 4F              MOV   C,A      ;MAKE CARRY BIT THE SOD
                                   ;ENABLE
44D 3E 01           MVI   A,01H    ;
44F 1F              RAR
450 1F              RAR
451 CD 30 04        CALL  BURST
454 AF              XRA   A        ;CLEAR A
455 CD 30 04        CALL  BURST
458 05              DCR   B        ;DECREMENT BIT COUNT
459 C2 44 04        JNZ   C01      ;REPEAT UNTIL ALL BITS
                                   ;IN THE BYTE ARE RECORDED
45C D1              POP   D        ;RECOVER D STATUS FROM
                                   ;STACK
45D FB              EI             ;ENABLE INTERRUPTS
45E C9              RET            ;RETURN
                                   ;END OF CASO SUBROUTINE


470 21 00 08 FLASH: LXI   H,PORTO  ;SET PORT POINTER TO
                                   ;PORT 0, ACTL
473 36 FF           MVI   M, FFH   ;TURN ON STATUS LAMPS
475 1E 0F     CONT: MVI   E, 0FH   ;INITIALIZE E COUNTER
477 16 FF           MVI   D, FFH   ;INITIALIZE D COUNTER
479 0E FF           MVI   C, FFH   ;INITIALIZE C COUNTER
47B 0D        DCC:  DCR   C        ;COUNT C
47C C2 7B 04        JNZ   DCC      ;IF NOT ZERO, JUMP BACK
                                   ;TO "COUNT C"
47F 0D              DCR   D        ;COUNT D
480 C2 7B 04        JNZ   DCC      ;IF NOT ZERO, JUMP BACK TO
                                   ;"COUNT C"
483 1D              DCR   E        ;ELSE COUNT E
484 C2 7B 04        JNZ   DCC      ;IF NOT ZERO, JUMP BACK TO
                                   ;"COUNT C"
487 EE 0F           XRI   0FH      ;ELSE COMPLEMENT
                                       STATUS LAMPS
489 C3 7B 04        JMP   DCC      ;AND RESTART COUNTING
                                       CYCLE
                                   ;END OF FLASH SUBROUTINE
```

Cassette Playback—ROM Version

```
4A0 21  00 00 PLBK:   LXI    H,0000H  ;INITIALIZE MEMORY POINTER
4A3 0E  FA            MVI    C,FA     ;START CHECKING FOR
                                          LEADER
4A5 CD C0 04  PB1:    CALL   BITIN
4A8 D2 A0 04          JNC    PLBK     ;CONTINUE LOOKING FOR
                                      ;LEADER ON THE TAPE
4AB 0D                DCR    C
4AC C2  A5 04         JNZ    PB1      ;WAIT FOR END
                                          OF LEADER
4AF CD  D0 04 PB2:    CALL   CASI     ;COMPILE THE BYTE TO BE
                                      ;ENTERED
4B2 71                MOV    M,C      ;STORE THE BYTE
4B3 2C                INR    L        ;INCREMENT MEMORY
                                          POINTER
4B4 C2  AF 04         JNZ    PB2      ;
4B7 24                INR    H
4B8 C2  AF 04         JNZ    PB2
4BB CD  70 04         CALL   FLASH    ;CALL FLASH.
                                          END OF PLBK
4C0 1E  16    BITIN:  MVI    E,16     ;
4C2 1D        BIT1    DCR    E
4C3 C2  C2 04         JNZ    BIT1
4C6 20                RIM             ;SAMPLE SID
4C7 17                RAL             ;MOVE BIT INTO
                                          CY POSITION
4C8 C9                RET             ;RETURN
                                      ;END OF BITIN SUBROUTINE

4D0 06  09    CASI:   MVI    B,09H    ;
4D2 16  00    TI1:    MVI    D,00H
4D4 15        TI2:    DCR    D
4D5 CD  C0 04         CALL   BITIN
4D8 DA  D4 04         JC     TI2
4DB 14        TI3:    INR    D
4DC CD  CO 04         CALL   BITIN
4DF D2  DB 04         JNC    TI3
4E2 7A                MOV    A,D
4E3 17                RAL
4E4 79                MOV    A,C
4E5 1F                RAR
4E6 4F                MOV    C,A
4E7 05                DCR    B
4E8 C2  D2 04         JNZ    TI1
4EB C9                RET             ;RETURN. END OF CASI
```

Expanding the Rodney System

The environmental inputs and action outputs on your Rodney system have the capacity for working with 16 bits of information. The main-memory system is likewise outfitted for 16 bits of environmental addressing. The programs utilized in Chapters 10 through 13, however, only call upon the lower 8 bits in each case.

It might appear, in other words, that you are using only half the system's full capability. That isn't quite the case. With an 8-bit system, you are working with 256 possible environmental conditions, action combinations, and main-memory addressing. If you move to a 16-bit system, the combinations reach nearly 65,000 in each case.

So, you are really only using a tiny fraction of Rodney's potential. You can stick on moveable arms and fingers, add 3-D sensing "eyeballs" and maybe even make him talk. It's all possible if you are clever enough to figure it out.

None of the work described thus far uses ENVH, ACTH, and MMAH (Ports 1 and 5 respectively). This is neither the time nor the place to embark upon a detailed set of instructions concerning the application of these unused ports. The time isn't right, because there is a need for more serious investigation into the properties of Rodney-class machine intelligence. This isn't the place to show more circuitry, even if we ignore further advances in machine intelligence—this book would simply be too large to be practical.

So let's assume by now that you are something of a Rodney expert. If you've been following the theories and instructions carefully, you probably want to try some of your own ideas anyway. Have a ball—it's all yours.

I will suggest a couple of simple add-on circuits that use the upper bytes in the system. These are only suggestions, however; do what you want with them.

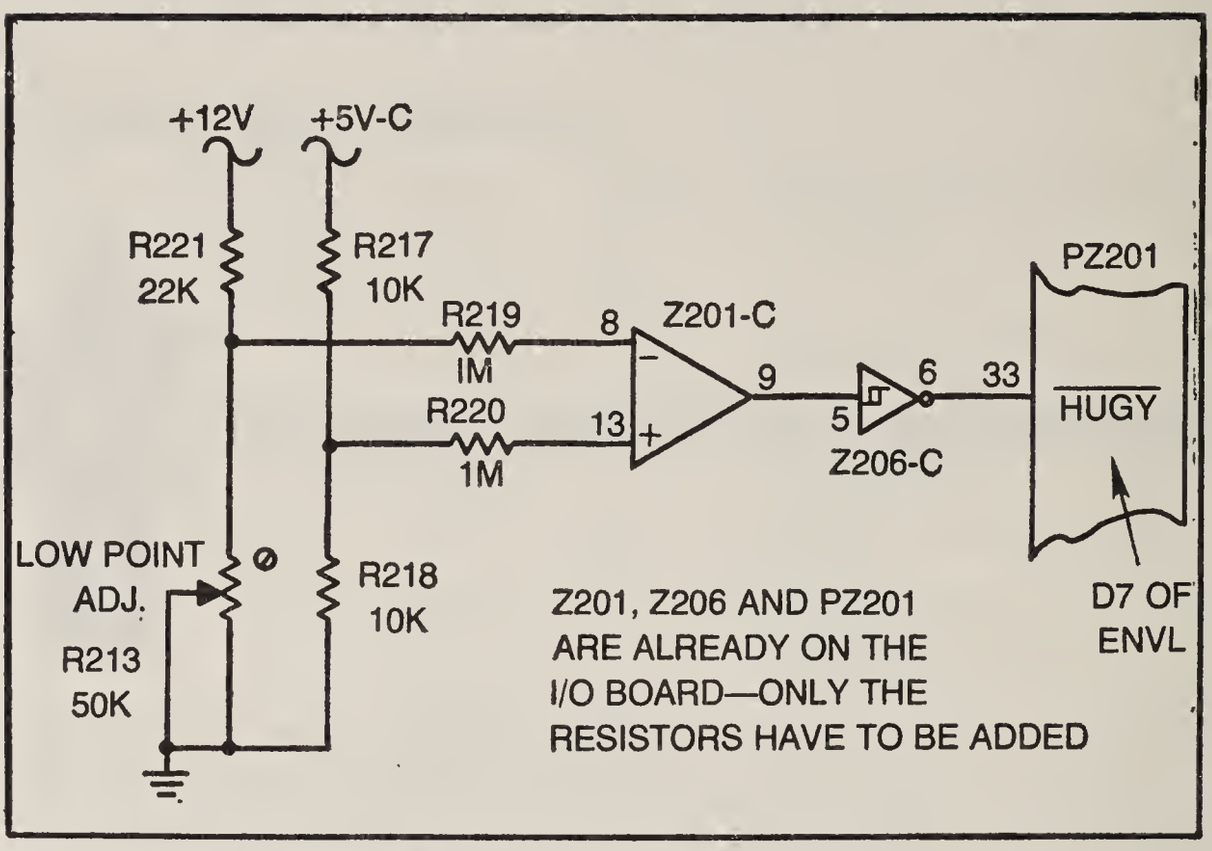


Fig. 14-1. A low-battery voltage sensing circuit.

LOW-BATTERY SENSING CIRCUIT

It would be nice if Rodney had some way to sense a low battery condition. A low-voltage sensing circuit is shown in Fig. 14-1. The nice thing about this particular circuit is that it uses portions of ICs already in use on the I/O board. All you have to add is a few resistors and a potentiometer.

The circuit is connected as a voltage comparator, one that compares a regulated 2.5V level with a sample of the battery vol-

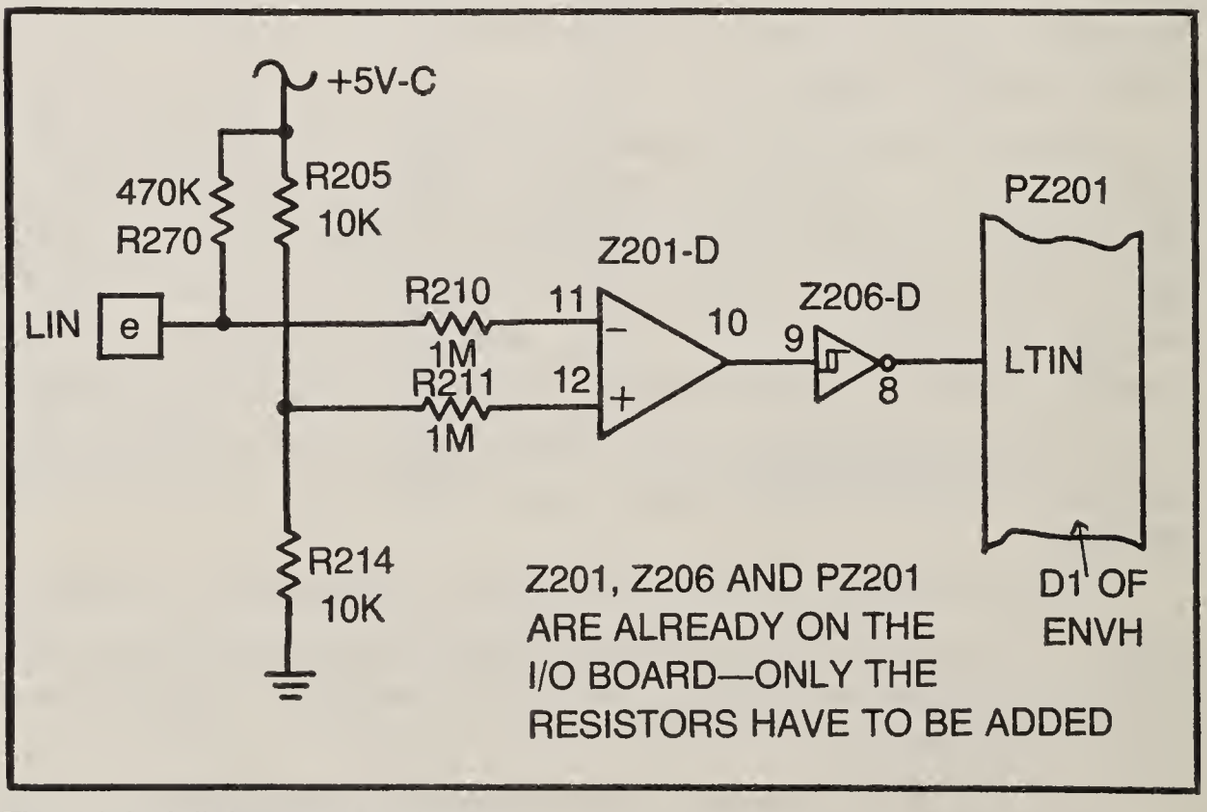


Fig. 14-2. A light-sensing circuit.

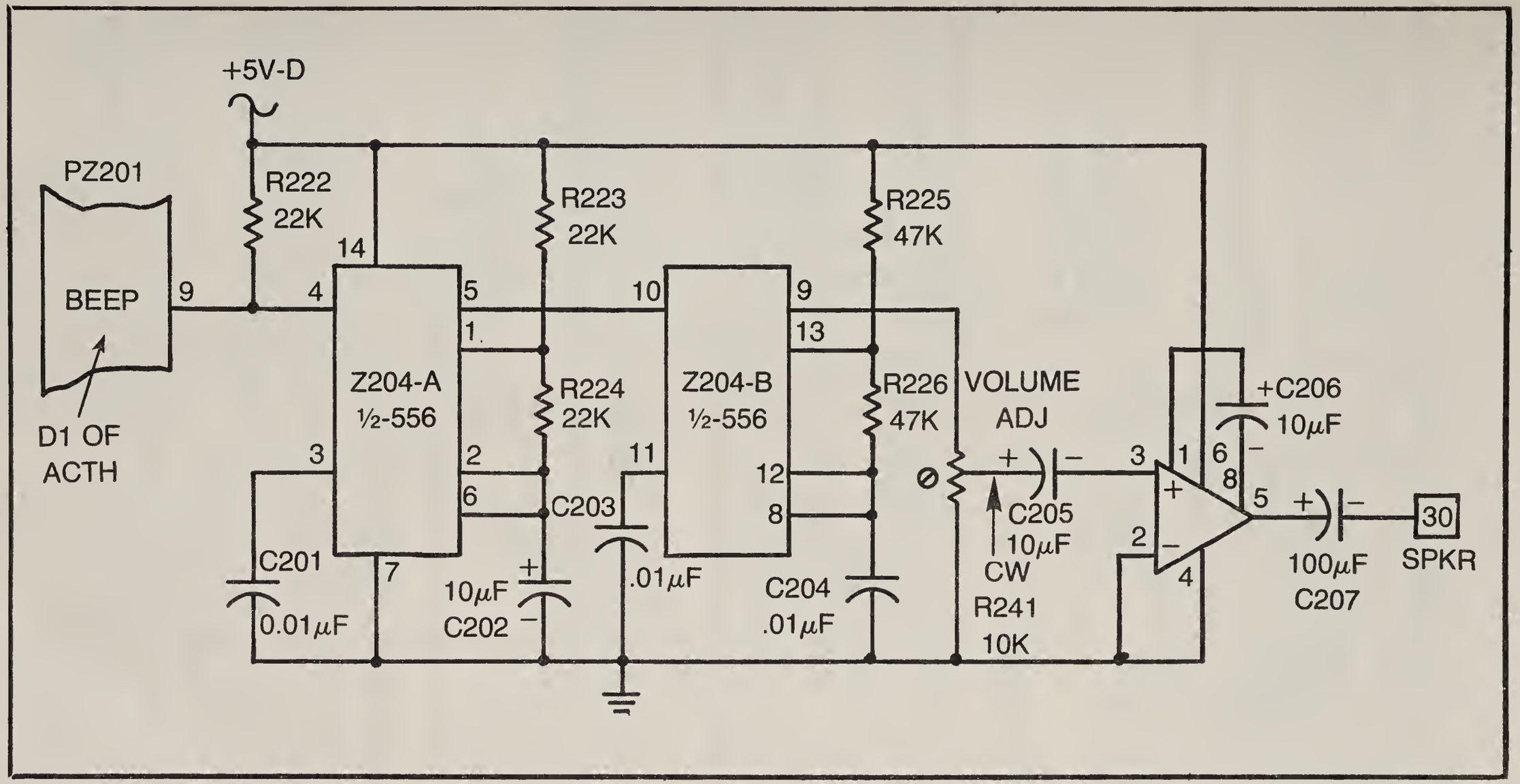


Fig. 14-3. A beep-tone generator circuit.

tage. The LOW POINT ADJ. pot should be adjusted so that the pin-9 output of Z201-C makes a transition from logic 0 to 1 when the battery voltage drops below 8V. The Schmitt-trigger inverter, Z206-C, inverts the phase of the signal and assures a clean, TTL-compatible HUNGRY alarm.

Applied to pin 33 of PZ201, this $\overline{\text{HUGY}}$ alarm becomes D7 of ENVL. The following program sequence thus gives you access to the low-voltage alarm:

```
aaa 3A  00 08        LDA     PORT0   ;FETCH ENVL
aaa E6  08           ANI     08H     ;ISOLATE HUNGRY BIT
                                     ;IF HUNGRY, Z
                                     ;IF NOT HUNGRY, NZ
```

GIVING RODNEY AN "EYE"

It is possible to make Rodney responsive to an external light source by adding a phototransistor and suitable interfacing circuit. The circuit is actually identical to the ones used in the motor motion-sensing circuit.

The amplifier circuit in Fig. 14-2 picks up the signal from the collector of a phototransistor at pin e on the edge-card assembly. The signal is amplified by Z201-D (a device already mounted on your I/O board), "cleaned up" by Z206-D, and supplied to the bus system as D1 of ENVH.

You can compose software to "tune" the circuit for certain flashing-light frequencies or simply to look for light/no-light conditions. The following program sequence places this LTIN bit into the accumulator of the microprocessor:

```
aaa 3A  01 08        LDA     PORT1   ;FETCH ENVH
aaa E6  02           ANI     02H     ;ISOLATE LTIN BIT
```

What you do with it after that is up to you.

GIVING RODNEY A "BEEP-BEEP" VOICE

The circuit in Fig. 4-3 is a simple audio tone generator which produces a simple beeping signal. The circuit is silenced as long as pin 4 of Z204-A is held at logic 0. Allowing this point to be pulled up to logic 1 starts the beeping tone.

The enabling signal for this circuit comes from bit D1 of ACTH—it is considered a valid *action* mode. The following program sequence causes the circuit to generate its beep tone, assuming of course the SPKR output is connected to an ordinary 8-ohm loudspeaker:

```
aaa 3A  01 08        LDA     PORT1   ;FETCH ENVH
aaa F6  02           ORI     02H     ;SET "BEEP" BIT TO 1
aaa 32  01 08        STA     PORT1   ;OUTPUT AS ACTH
```

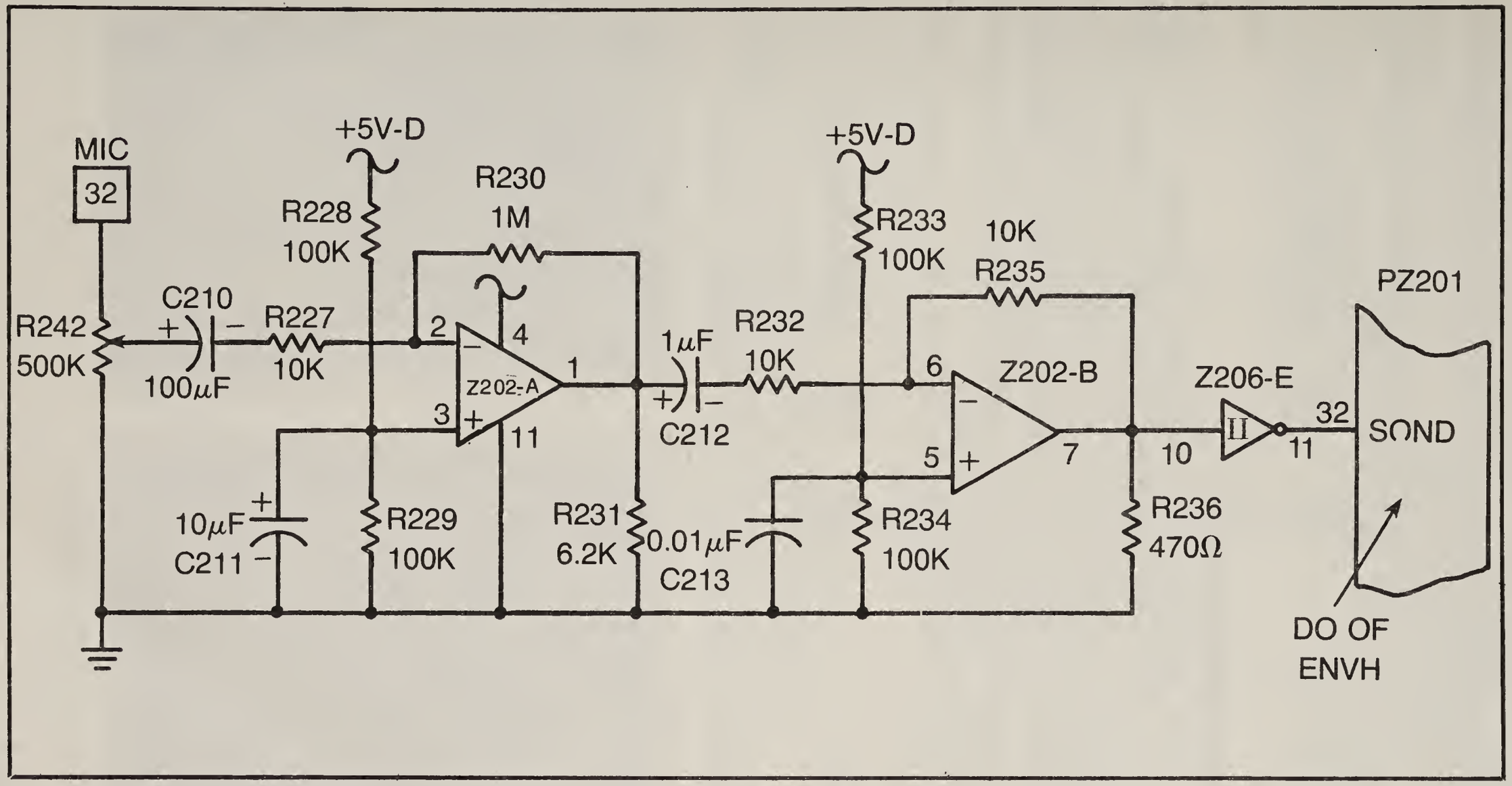


Fig. 14-4. An audio-sensing circuit.

The first operation fetches the present ACTH status and the second step sets BEEP to logic 1 without affecting the other bits in ACTH. The third instruction outputs the revised ACTH byte to output Port 1 and starts the beeping sound.

The next sequence turns off the beeping sound without affecting the status of the other bits in ACTH:

```
aaa 3A  01 08        LDA     PORT1   ;FETCH ENVH
aaa E6  FD           ANI     FDH     ;SET "BEEP" BIT TO 0
aaa 32  01 08        STA     PORT1   ;OUTPUT AS ACTH
```

GIVING RODNEY AN "EAR"

It is also possible to make Rodney responsive to an audio element of his environment. The circuit in Fig. 4-4 is simply an audio amplifier that picks up low-level audio signals from a microphone, amplifies them in Z202, and "cleans up" the result in Z206-E before making it D0 of ENVH. The volume control, R242, is an important part of this circuit because you probably won't want Rodney responding to the sound of his own motors.

The following program illustrates one way to get access to the SOND bit:

```
aaa 3A  01 08        LDA     PORT1   ;FETCH ENVH
aaa E6  01           ANI     01H     ;ISOLATE SOND BIT
```

After isolating the SOND bit in the accumulator, you can work out some software that makes the system respond to certain frequencies. Again, there are a lot of interesting possibilities.

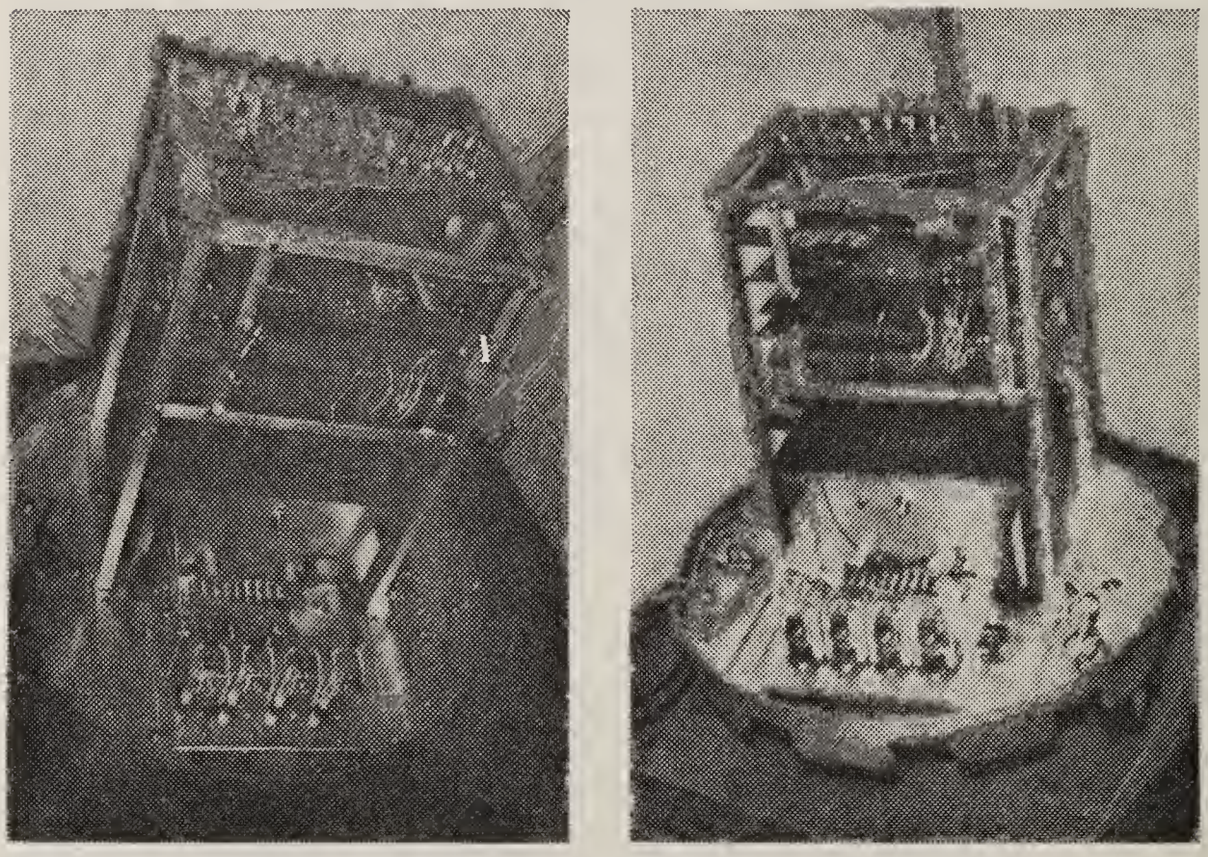


Fig. 14-5. Two views of Rodney.

ACCESSING A 2-BYTE MAIN-MEMORY ADDRESS

As you expand the environment sensing capability of your Rodney machine, you will also want to make corresponding expansions of the main-memory addressing. Rodney, as already described in this book, has a full 2-byte main-memory address. To this point, we have only used the lower byte. The following programs suggest one approach to addressing a 2-byte main memory and either loading or reading a 1-byte data word.

```
aaa 21   04 08      LXI     H,PORT4;SET PORT POINTER TO
                                   ;MMAL
aaa 3E   mm         MVI     A, mm  ;LOAD LOWER-BYTE ADDR
                                   ;INTO REGISTER A
aaa 77              MOV     M, A   ;LOAD LOWER-BYTE ADDR
                                   ;INTO MMAL
aaa 2C              INR     L      ;SET PORT POINTER TO
                                   ;MMAH, PORT 5
aaa 3E   nn         MVI     A, nn  ;LOAD UPPER-BYTE ADDR
                                   ;INTO REGISTER A
aaa 77              MOV     M, A   ;LOAD UPPER-BYTE ADDR
                                   ;INTO MMAH
aaa 2C              INR     L      ;SET PORT POINTER TO
                                    MMDL—
                                   ;PORT 6

aaa 3E   dd         MVI     A,dd    ;LOAD DATA INTO
                                    REGISTER A
                    MOV     M, A    ;LOAD DATA INTO
                                    MAIN MEMORY
aaa 2D              DCR     L
aaa 2D              DCR     L       ;RETURN PORT POINTER
                                     TO
                                    ;PORT 4
```

The foregoing is a procedure for storing new data (dd) into some main-memory address nnmm. The last two steps in the program merely reset the port pointer back to its original Port-4 level. Storing a new byte of data in this memory scheme uses much the same procedure. MVI A, dd is omitted, however, and the MOV M,A step that follows is replaced with MOV A,M.

Index