NEUTRINO PHYSICS - THE LINK BETWEEN THE MICROCOSMOS AND THE MACROCOSMOS, A STUDY IN TWO PARTS: (1) THEORETICAL - A LOOK AT THE TAU NEUTRINO MASS AND OTHER QUANTUM ELECTRODYNAMICAL EFFECTS IN THIRD FAMILY LEPTON INTERACTIONS AND (2) EXPERIMENTAL - UNDERWATER ASTRONOMY IN HAWAI'I, THE SHORT PROTOTYPE STRING OF THE DEEP UNDERWATER MUON AND NEUTRINO DETECTOR PROJECT

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF
THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSPHY

IN PHYSICS

DECEMBER 1989

By

John Freeman Babson

Dissertation Committee:

John G. Learned, Chairman
Charles Hayes
Michael D. Jones
Sandip Pakvasa
Lawrence J. Wallen

ACKNOWLEDGEMENTS

Starting out initially as a graduate student in the Institute for Astronomy, and from that experience learning lessons by no means limited to astronomy, then moving on to the High Energy Physics program, working first in the theoretical domain of electro-weak physics, and finally being intimately involved in the Short Prototype String experiment of the DUMAND project has been a long journey. Now that it is almost over, looking back at it, I can honestly say that any exercise as extensive as this is not done in a vacuum. Over the years there have been many who have contributed both directly and indirectly, materially and immaterially, to my efforts in graduate school. Time and room do not permit me here to adequately acknowledge each individual in terms of their contributions. Instead, for the most part, I shall endeavor to list them by name and approximate affiliation, knowing that deep inside of each individual is that part of them which they have contributed to this effort that I shall always appreciate. I know that I am the person I am today in part because I have learned something from each one of these individuals.

Institute for Astronomy, University of Hawai'i at Manoa:

Dana Backman, Erik ("the Infrared") Becklin, Charley Beerman, Jeff F. Bell, ("Raggedy") Ann Boesgaard, Bob Brown, Tim Brown, H. Melvin Dyke, Mike Gaffy, John C. Good, John Hamilton, Charles Heasley, ("Maximum") John T. Jefferies, Jeff Morgan, Norman G. Purves, Robert Routsalainen, Brent Tully, and Gareth Wynn-Williams

Diamond Sangha, Koko An Zendo:

Anne Aitken, Robert Aitken Roshi, Linda Brown, Duke Choy, Mary Choy, Christopher Fang, P. Nelson Foster, Francis Haar, Michelle Hill, Deborah Hopkinson, Patrick Jijaku, Michael Kieran, Karen Loftstrum, Marion Morgan, Loren Rivers, Clifton W. Royston, Greg Shepard, George Steele, Don Stoddard, Vicky Stoddard, Bobby Swartz, John Tarrant, Andrew Thomas, Teresa Vast, and the late Yamada Koun Roshi

Physics Department, University of Hawai'i at Manoa:

Kaladi S. Babu, Ralph Becker-Szendy, Douglas Brandt, Sitaram Byahut, Caroline Chong, Peter Crooker, Steve Dye, John Henry Flack, Calvin Fujitani, Chris Fujiyoshi, Jim Gaines, George Gerhab, Peter Wellington Gorham, Jeremy W. Gorman Jr., Ward Hayward, Burton Henke, Donald Hoffman, the late Jeff Hoover, Jacob Hudson, Diane Ibaraki, Vivek Jain, Cheryl Kamiyama, Kathleen Kaya, In-Sook Lee, Grace Lim, Sheila Mae Lopes, Jack MacMillan, Melvin Matsunaga, Teri Mimori, Wesley Mizuno, Paul Montecastle, Jon P. Okada, Devi D. Paudyal, Rupert C. Perera, Kandatege ("Primo") Premaratne, Antonio Querubin, the late John Jun Sakurai, Mark Schindler, Walter Simmons, Walter Steiger, Glenna Sumiye, Robert Svoboda, Hirotaka Sugiwara, Tina ("Tigger") Tanaka, Xeres Tata, Roy Tom, San-Fu Tuan, and David Yount

Friends:

Nancy Aleck, Olga Alexandrov, Oi-Man Chan, Man-Tin (Clement, Tim) Cheung, Yick-Wa (Grace, Ewa) Cheung, Hiromi Otsuka Chiba, Peter and Kae Hyung Chin, Dick Flagg, Alberta Freidus, Danielle Irene (Chin) So-mai Ho, Marian Ho, Fran Johnson, the late Trudy Kim, Wayne Kamimoto, Peter Kwong, Bonnie Lau, Herbert (Kwan) Lee, Yon-Un Lee, Ruey-Fen Lin, Paul J. Mareno, Guy and Denise Mazzanti, Don Micheal, Marcia Nelson, Edith Ngai, Francis Duk-Wing Pang, Richard Rose, and Karen Taira

Landlords and housemates:

Nancy R. Clark, Sue Congdon, Chuck Mueller, Mora O'Connor, Florence Tyau, and Bernard T.S. ("Tough Shit") Wong

DUMAND:

Grant Blackington, Hugh Brandner, Cpt. Clary and the crew of the SSP Kaimalino, John Clem, Ray Cote, Tom Daniels (Hawaii Natural Energy Laboratory), Jim Elliot, Gary and Sonya Friedhoffer (Camp Timberline), Jim Gaidos, Peter Grieder, Dave Harris, Matti Jaworski, Shigenobu Matsuno, Bob Mitiguy, Yoshiko Miyakoshi, Dan O'Connor, Y. Ohashi, A. Okada, Vincent Z. Peterson, Fred Reines, Arthur Roberts, Charley Roos, Mark Rosen, Hank Sobel, Victor Stenger, Med Webster, George Wilkens, and Chuck Wilson as well as the Kaena Point USAF Communications Facility and the Hawaiian Telephone Company

Family:

Tom Freeman Babson, June I. R. Babson, the late Norman I. Freeman, Dorothy B. Freeman, Sara J. Freeman, Charles D. Freeman, Marge S. Freeman, David H. Freeman, Jean Freeman, Jim Freeman, Judy Freeman, Beth I. Freeman

Inspirational folks:

Mitsui Aoki, Herbert B. Falkenstein, John M. Koller, William S. Merlin, and Gary Snyder

Dissertation Committee:

Charles Hayes, Micheal D. Jones, Sandip Pakvasa, and Lawence J. Wallen

Graduate Advisors:

Special mention must be made of my graduate advisors, for during this long siege I have been blessed with three. I started with Carl Pilcher in astronomy. As a fellow chemist, he was very supportive of me in the world of astronomy and for that I am greatly appreciative. Both of us have come a long ways since then and I suspect that life has taught us a lot. I know that we share the understanding that in this threatened world of ours there is much more to it than just pure science research. The need for human beings to make true contact with one another is imperative. Ernest Ma was my advisor in theoretical high energy physics and a true mentor. From him I learned not to be afraid or embarrassed of my intuition but rather to give it free reign in probing the unknown. He taught me to look beyond a mere formalism and instead to focus on the heart of a problem, often by looking at it perhaps sideways rather than straight on. He is a saint for he supported me through the twin traumas of Yick-Wa's breast cancer and my Mother's stroke and for this I shall never be able to find words of appreciation deep enough. Lastly, there is John Gregory Learned, who has been my advisor for the last six years while I have labored on the Short Prototype String of the DUMAND project. The lessons I have learned from him are perhaps some of the most profound of my life. It is through him that I can honestly say that I have finally been able to find the means by which to articulate much of my life's earlier experiences and as such have been able to come to a deeper and more fundamental experience of them. Thank-you John.

Micheal Jones read each and every page of the manuscript and made many excellent and detailed comments which I found extremely helpful. I am convinced that as a result of his effort the clarity of presentation of this dissertation was greatly improved. He certainly deserves the best spelling award from among the members of my committee. Sandip Pakvasa also made many suggestions to the theoretical portions of this dissertation which resulted in much improvement. My sincere thanks to both of these gentlemen.

There is a final acknowledgement that must be made. I begin with thanking the trees for having given their all. Much paper has been consumed in the process. I hope my use can be justified in the spirit of man playing the role of being one way for nature to come to know itself. Also, there is a mountain named Mauna Kea (Hawaiian for "white mountain") which makes up half of the island of Hawai'i. I realize that I am one of the very few human beings privileged to have done research at the very top of it (at an altitude of 14,000 feet with the University of Hawai'i 88" optical / infrared telescope observing faint dwarf galaxies) and the very bottom of it (at a depth of three miles with the Short Prototype String seeking cosmic ray muons and neutrinos). I feel a particular sense of intimacy with Mauna Kea that is very hard to describe in words and simply wish to acknowledge its existence and to thank Pele for allowing me to play there. In the long run, the only true teacher is Nature and it seems to be that the peculiar problem of our age is how often we neglect this fact.

Finally, in terms of shear inspiration (the "breathing in", life itself), I must acknowledge and thank Fung-Chu (Candy) Ho. In the most profound sense she takes me beyond my meager self and opens my eyes to the larger evolving world. Whether she understands this or not, outside of my parents and my childhood, she has taught me more about the world than anyone else. It is through her that I have been able to look back at things and finally come to understand much that I have experienced and yet never properly digested. She forces me to be honest. She shows me that balance in life is both possible and essential. To say that she is me and that I am her is an understatement. Her good counsel is always welcome. I love you Fung-Chu! Thank-you!!

I hereby dedicate this work to the living memories of my Father,

John Capron Babson (1902 - 1971)

who as a human becoming and engineer's engineer taught me the importance and delight at knowing the physical world, before I even learned to talk, and of my Mother,

Leona Freeman Babson (1906 - 1989)

who as a human becoming and teacher's teacher taught me the importance of the human element right up to the moment of her recent death. To paraphrase Sir Issac Newton, standing upon the shoulder's of such giants I can not but help see beyond them. May the rest of my life, which I hereby dedicate, be worthy of the legacy of my many teachers.

ABSTRACT

Until recently, the subject of the very largest dimensions in the Universe, namely Cosmology, and the subject of the very smallest dimensions in the Universe, namely Particle Physics, were considered to have very little to do with one another. In terms of observation, the former was the province of the extragalactic Astronomer and the latter the province of the particle Physicist. The decade of the 1980's has forever changed that. Such a separation is now known to be an artificial construct at best. An example is the verification with SN1987a that supernovas are powered by a blast of neutrinos! The possibility of dark matter pervading the whole Universe may ultimately result in a mass count that could close the Universe. The lowly neutrinos, should they indeed have some mass, are potentially so plentiful that they may do it alone. Fundamental questions are being answered in Cosmology by way of Particle Physics and in Particle Physics by way of Cosmology.

In a small way, this dissertation reflects this softening of the boundaries. It does so in two ways. First, there is a theoretical investigation into some aspects of generational universality. The consequences of a third lepton, namely the tauon, and an associated tau neutrino, are explored in terms of phenomenology that may shed insight into questions of neutrino mass and increased symmetry at higher energies. Second, there is an experimental investigation in the form of constructing and operating the first stage of the DUMAND (Deep Underwater Muon and Neutrino Detection) project which was a ship suspended muon and neutrino telescope called the SPS (Short Prototype String). This detector is of the water Cherenkov type and is the first time such an instrument has been successfully built and tested for use in the ocean.

It is very likely that such detectors will be the common place future of the experimental high energy physics community since accelerators would have to approach in size the diameter of the Earth in order to move just a

few orders of magnitude higher in energy. Like the first cyclotron, as in any such first time exploration, much in the way of new technological integration was necessary and my contributions to this end are documented here.

Chapter 1 begins the first of two parts to this dissertation and is a brief introduction to the Microcosmos and some of the outstanding problems in the exploration of the realm of the very small. The theoretical approach taken to look into third generation phenomenology was to consider the tauon, generated in an electron-positron colliding beam, as it decayed into an observable meson and a tau neutrino. Chapter 2 reviews the state of the art in terms of what was measured at the time of this investigation. Chapter 3 describes the calculations I carried out on the decay-product correlation of $\tau^+\tau^-$ into observable mesons. The correlation angle is then associated with such phenomena as the mass of the tau neutrino and the possibility of a higher energy interaction including some V+A mixed in with the V-A.

Chapter 4 begins the second part of this dissertation and is a brief introduction to the Macrocosmos and some of the outstanding problems in the exploration of the realm of the very big. Chapter 5 reviews the state of the art in terms of experimental approaches taken up to the development of the SPS. Chapter 6 is the first of five chapters devoted to the detailed documentation of parts of the SPS and its technology integration that I designed, prototyped, and debugged. This chapter describes the command and control communications system used to control the various modules (instruments) found on the string. Chapter 7 describes the design of the microcontroller circuits used in the optical and calibration modules. Chapter 8 describes the design of the microcontroller circuit used in the control of power distribution throughout the SPS. The microcontroller used in the above had a lot of problems associated with it, the most outstanding being its very crude instruction set. Chapter 9 details the UHPS (Underwater Hawai'i Programming System) language which was developed to overcome this limitation. Chapter 10 details the design, both hardware and software, used to control the SBC (String Bottom Controller) of the SPS. Software details in terms of program listings make up several of the appendices. Finally, Chapter 11 is an analysis of SPS data in terms of ascertaining a purely statistically based downward traveling muon rate at a depth of 4.0 Km. It is found to be $(2.06 \pm 0.68) \times 10^{-2}$ Hz. Assuming a muon flux of $7 \times 10^{-5}\,\mathrm{m^{-2}\,s^{-1}\,sr^{-1}}$ (at 4.0 Km, following Kobayakawa) this corresponds to an effective area of $A_{eff} = 3 \pm 1 \times 10^{2}\,\mathrm{m^2}$. Additionally, a determination of the power index (n) of the cosine of the zenith angle of the downward traveling muons has been made and found to be $n = 5.3$ which is consistent with previously reported results from deep mine experiments.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF ILLUSTRATIONS

LIST OF ABBREVIATIONS AND SYMBOLS

ACRONYMS

Acronym	Definition
ABX	computer register set model (named after its registers)
ADC	Analog to Digital Converter
ADC	Analog to Digital Control (multi-channel)
ADCLK	A to D CLocK
AGC	Automatic Gain Control
ALE	Address Latch Enable
ALU	Arithmetic Logic Unit
A to D	Analog to Digital
ASCI	asynchronous serial communications interface
ASCII	American Standard Code for Information Interchange
BIOS	Basic I/O System
CC	Condition Codes
CERN	Center European Radiation Nuclear (French acroynm so in English the word order makes sense in reverse)
Cn	capacitor n
CMOS	complimentary metal oxide silicon logic
COMP-IN	COMParison INput
CP/M	Control Program / Microcomputers
CPU	central processor unit
CSIO	clocked serial I/O port
CWI	Case-Witwatersrand-Irvine (experimental collaboration)
DAA	Data Access Arrangement
DAC	Digital to Analog converter
DACPWR	DAC PoWeR
DAEn	Digtal to Analog Enable line n
DATAST	DATA STrobe
DCD	data carrier detect
DDT	Dynamic Debugging Tool

DMAC	direct memory access controller
dork	the phallic shaped extension of the calibration modules used to convey the generated light pulses to a scintillator ball out of the axis of the string of optical modules in the SPS
DREAD	data read
D to A	Digital to Analog
DUMAND	Deep Underwater Muon and Neutrino Detection
EA	External Address
ECL	Emitter Coupled Logic
EN	ENable input
EOC	end of conversion
EXP SEL	EXPansion SELect
FORTRAN	FORmula TRANslation (name of computer language)
FM	frequency modulation
FSK	frequency shift keying
GIM	the GIM mechanism named for S. Glashow, J. Iliopoulos, and L. Maiani
GUTS	Grand Unified Theories
HEPG	High Energy Physics Group
ICE	in-circuit emulater
IMB	Irvine-Michigan-Brookhaven (experimental collaboration)
INTn	INTerrupt line n
I/O	Input / Output
IOEN	I/O Enable
KEK	The Japanese national accelerator center
KGF	Kolar-Gold Field (experimental collaboration)
LAN	local area network
LATn	LATch enable n
LASTAT	latch status
MASTST	MASTer reSeT
MCLK	Modem CLocK
MERLIN	SBC control program

MIDS	Module IDentification System
MDS	Microcontroller Development System
MMU	memory management unit
MPL	Macro Programming Language
MUX-OUT	MUltipleXer OUTput
OE	Output Enable
o.s.t.	one sided triggering
IBM	International Business Machines
PC	Program Counter
PDK	Proton Decay experiment (another term for the IMB)
PL/I	Programming Language one (name of computer language)
PMT	photomultiplier
PRT	programmable reload timer
PSEN	Program Sense ENable
PWR	PoWeR module control program
QCD	quantum chromodynamics
Q.E.D.	quantum electrodynamics
RAM	Random Access Memory
RATECMP	RATE CoMPare - program for unpacking and reading the raw scalar data from the SPS data files
RATEHST	RATE HiSTogram - program for calculating the expected random 5-fold or greater aggregate coincidence rate r and compares it to the actual 5-fold or greater aggregate coincidence rate R
RATFOR	RATionalized FORtran (name of computer language)
RD	ReaD control
REFn	REFerence of voltage n
ROM	Read Only Memory
ROMAD	ROM ADdress
Rn	resistor n
RXD	Receive (digital) Data (as opposed to analog)
SLAC	Stanford Linear Accelerator Center

SLRNK+	SuperLinker Plus
SBC	String Bottom Controller
SOM	String Optical Module program
SP	Stack Pointer
SPS	Short Prototype String of the DUMAND Project
SU(5)	Special Unitary symmetry of order 5
TOS	top of stack
TPA	Transient Program Area
ZCPR3	Zilog CP/M Replacement number 3
ZRDOS	Zilog Replacement Disk Operation System
TTL	transistor-transistor logic
X	indeX register
XBUS	external buss
TXD	transmitter data line
TXD	Transmit (digital) Data (as opposed to analog)
UART	Universal Asynchronous (serial data) Transmiter / Receiver
UHPS	Underwater Hawai'i Programming System
UV	ultraviolet
V - A	vector minus pseudovector or "left" handed interactions
V + A	vector plus pseudovector or "right" handed interactions
VH	Voltage High (logic sense reference signal)
$\underline{\text{WR}}$	WRite control
WWV	call sign for universal time clock run by the U.S. National Bureau of Standards
XTALn	crysTAL line n
Yn	crystal n
ZCPR3	Zilog CP/M Replacement number 3
ZRDOS	Zilog Replacement Disk Operation System

SYMBOLS

Symbol	Definition
A	isotropic part
B	anisotropic part
c	the absolute speed of light in a vacuum
c	the charmed quark, analogous to the up quark
$d\sigma(\theta_{corr}) / d\Omega$	overall correlation angle dependent cross section
$d\,\sigma(s_1, s_2) / d\,\Omega$	the spin dependent differential cross section
$d\Gamma_1 / d\Omega_1$, $d\Gamma_2 / d\Omega_2$	the spin dependent τ decay distributions
E	the energy operator
E	energy of $e^+(e^-)$ beam
E_{cm}	center of mass (momentum) energy
$g_w = \sqrt{4\pi\alpha_w}$	= the "weak coupling constant"
H	number of hits (events)
$\hbar$	the quantum Planck constant $= h/2\pi$
$H(\mathbf{p},\mathbf{r})$	the classical hamiltonian
$h(s)$	numbers of "hits" (events)
$\mathbf{I}$	isospin vector
I_z	the total projection of a system of interacting particles
I_θ	the vertical intensity at a zenith angle θ
I_0	the vertical intensity at the zenith
J/Ψ	the double named charmonium particle
$\mathbf{L}$	angular momentum
$\mathbf{L}_e$	intrinsic angular momentum (spin) of electron
m_p	mass of the proton = 0.9383 GeV/c^2
m_n	mass of the neutron = 0.9396 GeV/c^2
m_τ	mass of the tauon = 1.784 GeV/c^2
m_π	mass of the pion = 0.140 GeV/c^2
$m(\nu_\tau)$	mass of the tauon neutrino

n	the power law exponent for the variation of the power of the cosine of the zenith angle of penitrating cosmic ray muons with respect the depth of measurement in the Earth
$\binom{n}{m}$	count (or weight) of the number of identical terms $p^m q^{n-m}$ making up the aggregate probability of an m-fold occurrence among n things
$n(\nu_\tau)$	number density of the tau neutrino in the Universe
n_c	(critical) energy density to close the Universe
p	individual probability
$\mathbf{p}$	the momentum operator
$\mathbf{p}$	linear momentum
P_π	the momentum of the composite pion
$P(m)$	aggregate probability for m things each with a common finite of occurrence tried n times
$\mathbf{r}$	radius
r	associated random rate within the coincidence window time
r	predicted aggregate rate [Hz]
R	real aggregate rate [Hz]
$r(z)$	radiative correction term
t	integration time [seconds]
u	the up quark
v	the speed of a particle
W^+,W^-	the charged massive gauge bosons of electroweak unification
z	electron energy relative to the maximum possible value of $E_e / E_e max$
Z^0	the neutral massive gauge boson of electroweak unification

α and β	constants such that $\alpha_i^2 = 1, \beta^2 = 1$ $(i = 1, 2, 3)$
α_e	the fine structure constant
β	velocity of $\tau^+(\tau^-)$
ε	measure of the degree of pureness of V-A (pure V-A $[\varepsilon = 0]$)
ε_i	statistical error
$\varepsilon(\omega)$	the dielectric constant of a material medium frequency ω
η	measure of anisotropy to isotropy = $(B\ /A)^2$ (note that for $\eta = 0, m_\nu = 0$)
θ	production angle
θ_c	the Cabbibo angle
θ_c	characteristic angle of emission of Cherenkov light
θ_{corr}	correlation angle
ρ	Michel parameter
τ	coincidence window time
$\tau(\tau_0)$	lifetime of the tauon
$\Psi(\mathbf{r},t)$	the wave function

PREFACE

The work contained in this dissertation, both theoretical and experimental, has been supported by and is an extension of the work done by the High Energy Physics Group (HEPG) of the Department of Physics and Astronomy of the University of Hawai'i at Manoa in Honolulu Hawai'i. For the most part, this has been funded through the ongoing contract that HEPG has with the United States Department of Energy. As noted in the acknowledgements section, many people, particularly those associated with HEPG have contributed towards the general advancement of knowledge that this dissertation represents. However, in the long run, this dissertation is a summary of the work that I did and as such the author is responsible for any shortcomings in the material contained herein.

The main text of this dissertation was set in New Century Schoolbook[1].

[1] as popularized in the Dick and Jane primer series

PART I – THEORETICAL CONSIDERATIONS

CHAPTER 1 – Introduction to the Microcosmos

This study is in two parts and is a reflection of the state of the art of high energy physics in the 1980's in that it is in this decade that the study of both the very small, in the form of elementary particle physics, and the very big, in the form of ultrarelativistic cosmic rays and their impact on cosmology, can be truly said to have come together. The first part deals with some aspects of theoretical electro-weak physics that I worked on in the early part of the decade. The second part reports the experimental work that I did in support of the Short Prototype String of the DUMAND (Deep Underwater Muon and Neutrino Detection) project. To start with the first part, a brief review of the history of electro-weak physics is in order.

1.1 Quantum Electrodynamics

By 1928, not very long after Schrodinger and Heisenberg came up with their different yet valid quantum mechanical formalisms in 1925, it was realized that for real problems dealing with the wave functions of atoms and nuclei that a relativistic form of quantum mechanics was needed. This need was fulfilled by P. A. M. Dirac.

1.1a Early Quantum Electrodynamics ala Dirac

One of the great triumphs of twentieth century physics has been to take the new theories of special relativity and quantum mechanics and in combining them be able to derive from deeper principles Maxwell's theory of electromagnetism. This theory is the first of the quantum field theories to explain one of the fundamental interactions (forces) of Nature. Thanks to the humor of R. P. Feynman it is known as Q.E.D. (quantum electrodynamics). The naive approach to combining quantum mechanics with special relativity would be to start with the Schrodinger equation

$$E\,\Psi = H\,(\mathrm{p},\mathrm{r})\,\Psi \qquad (1.1)$$

where E is the energy operator, $H(\mathbf{p},\mathbf{r})$ is the classical hamiltonian and $\mathbf{p}$ is the momentum operator. The function $\Psi(\mathbf{r},\mathrm{t})$ is the wave function for the particle whose probability density of finding the particle at point $\mathbf{r}$ at time t is $[\Psi(\mathbf{r},t)^2]$. One tries to go about this by substituting the relativistic hamiltonian

$$H = [(mc^2)^2 + \mathrm{p}^2c^2\,]^{1/2} \qquad (1.2)$$

This fails because E and $\mathbf{p}$ do not occur in the equation in a similar manner and thus the equation is not relativistically invariant. Another attempt is the Klein-Gordon equation

$$E^2\phi \;=\; [(mc^2)^2 + \mathrm{p}^2c^2]\phi \qquad (1.3)$$

which fails because ϕ can not be a wave function (the equation is second order and so future values of ϕ can not be determined without also its derivative with respect to time) and its density is not positive definite. Dirac's equation

$$E\,\Psi = [\beta mc^2 + \alpha\cdot\mathrm{p}\;c\,]\,\Psi \qquad \rho \;=\; \Psi^*\Psi \qquad (1.4a)$$

has a positive definite probability density ρ with α and β being constants such that

$$\alpha_i^2 = 1 \qquad \text{and} \qquad \beta^2 = 1 \qquad (i = 1, 2, 3) \qquad (1.4b)$$

and all of the other combinations are zero. This equation results in four eigenstates with two positive energy states with opposing spins and two negative energy states with opposing spins. The negative energy states are identified with the "anti-particle" of the positive energy state particle. Taking the arguments further, it turns out that there is a limited set of possible interactions (solutions to the Dirac equation) for the wavefunctions Ψ of the form:

$$\Psi'(x') \text{ op } \Psi'(x') \qquad (1.5)$$

where op stands for a relativistically invariant operator. These are the scalar, the vector, the antisymmetric tensor, the pseudovector, and the pseudoscalar solutions. In combination, these are referred to as the V - A (vector minus pseudovector or "left" handed) and the V + A ("right" handed) interactions.

1.1b Later Day Q.E.D. ala Feynman

Dirac's description, while it took the quantum theory of electrodynamics a long ways, was not completely satisfying. Strictly applying it, one would have an infinity of higher order terms based on the self interaction of the electromagnetic field. Thus, the theory accurately predicted many things such as the existence of anti-particles or the magnetic moment of the electron, but it did not result in real interaction solutions. By the late 1940's, R. P. Feynman and others found a way around this difficulty. They showed that because the parameter coupling of the field to matter is small compared to unity (actually the fine structure constant of 1/137), then small perturbations rapidly close the series. Thus, the infinity of higher order terms do not contribute anything. In fact, Feynman became famous for his introduction of a simple diagrammatic technique which allows one to visualize the contributions due to increasingly more complicated interactions without having to get immediately lost in a sea of mathematics where one could easily overlook a term that should be included for series out to any given order n.

Such theories are said to be locally gauge invariant. What this means is that the symmetry in the theory is such that the theory is invariant with respect to local phase rotations of the wavefunction. These are proportional to the electric charge of the associated particle if and only if the field has precisely the properties of the electromagnetic field of Maxwell. The choice of phase is conventional and not physical and thus independent of an arbitrary gauge of measurement.

An aspect that ought to be mentioned in passing, is that the photon, which is the propagator in QED theory, is massless. This means that QED is a quantum field theory whose propagator has an infinite range. This sort of thinking was what H. Yukawa had in mind when in 1935 he predicted that some kind of heavy mediating particle, the meson, was needed to mediate the (strong) nuclear force between nucleons inside an atomic nucleus since the range of this new force was known experimentally to be confined to the dimensions of the nucleus. In other words, in some sense, the range of a propagator is inversely proportional to its mass. Additionally, a theory in which the propagator is massless has a symmetry which is not broken. Finally, it should be pointed out that the gauge principle is a very deep principle, in contrast to a lot of ad-hoc things floating around in the world of quantum field theory, and as such takes on great importance in trying to develop theories for other interactions.

1.2 The Particle 'Zoo"

The term "elementary" when applied to particle physics must be applied with a healthy sense of skepticism. In the recent past, the known elementary particles would have been the molecules and later the atoms which make up the periodic table of the elements. More recently, the term could be applied to the nuclei of the atoms and even later to the nucleons (and mesons) that make up the nucleus. Today, the current picture is that even these elementary particles are made up of even more fundamental particles (quarks). A second point that is worth mentioning, before embarking on our adventure through the elementary particle zoo, for at times it is an important guide along the journey, is that there is a sort of meta-axiom which says that whatever is not explicitly forbidden must happen.

1.2a Quantum numbers

Elementary particles are completely (uniquely) described by the totality of their internal quantum numbers. Internal means that the quantum numbers refer to the particle itself in contrast to the classical notions of energy, momentum, and angular momentum which deal with the "external" context of space and time within which the particles may find themselves. Two concepts, namely mass and charge, are carried over from classical physics while the rest are purely quantum mechanical developments. What follows is a brief review of these internal quantum numbers.

Spin - spin is the "intrinsic angular momentum" of a particle. It is needed so that the total angular momentum in a particle interaction is conserved. It was first introduced with respect to the electron and found to have a value of

$$\mathbf{L}_e = \mathbf{r}_e \times \mathbf{p}_e \qquad (1.6a)$$

where the subscripts all refer to the electron (e) with angular momentum (L), radius (r), and linear momentum (p). Since the electron is a point-like particle with no radius, the usual interpretation of $\mathbf{L}_e$ as the angular momentum in configuration space makes no sense. Thus, the spin acts like or with the usual configuration space angular momentum to retain total angular momentum conservation but it is internal or intrinsic to the particle. Spin takes on values that are either half-integer (fermions like the electron) or integer (bosons) multiples of ($\hbar = h/2\pi$). These values are viewed to be the projections of a spin vector which rotates about a fixed origin. The number of possible spin states for a given spin L is given by

$$\text{number of states} = 2L + 1. \qquad (1.6b)$$

Spin exhibits the kinds of properties that one typically associates with quantum numbers, namely that they are intrinsic to the particle itself and that they take on discrete rather than continuous values. In this sense, the electric charge, a well known quantity from classical physics is also a quantum number.

Isospin - Heisenberg in 1932 made the observation that the proton and the neutron both have essentially the same mass (m_p = 938.3 MeV and m_n = 939.6 MeV). Thus, he speculated, that the two particles could be viewed in some sense as really being the same particle in two different states. The truly important observed difference being their electric charges (e_p = e and e_n = 0). The latter observation that the n-n, p-p, and especially n-p strong interactions are all of the same strength bare out this hypothesis. Basically, the assignment of isospin values, analogous to spin values, allows one to group together particles of almost identical masses but of different charges as different states of the same particle (singlets, doublets, triplets, etc.). What is conserved in strong interactions is the absolute value of the (self consistently assigned) isospin vector I and the total projection I_z of a system of interacting particles.

The isospin of a particle is completely "internal" in its action in the sense that it does not couple with anything other than the isospin of another particle. The abstract space in which this happens is termed the isospin space. This initial usefulness of isospin was as a quantum number conserved in strong interactions which summarized the spectrum of strongly interacting particles, known to have real radii (i.e. not to be point-like), hinted at these particles as being composed of something more fundamental.

Strangeness - in the early 1950's certain interactions that were not otherwise forbidden were never detected. In particular, the K-meson and the Λ-meson were never observed independently but the interaction

$$\pi^- + p \dashrightarrow \Lambda^0 + \pi^0 \qquad (1.7)$$

was. Thus, following the meta-axiom of what is not explicitly forbidden must be allowed, Gell-Mann and Nishijima (1952) proposed the adaptation of yet another quantum number, called strangeness, which in order to be conserved in interactions would exclude some otherwise possible interactions. Self consistent assignments were indeed found to be possible which thus were able to "explain away" this "strange" behavior on the part some elementary particles.

Color - as a quantity, the electric charge has two roles to play. One role is that of a quantum number where it takes on only discrete values (whether the charge quantum is considered to be either e_p or $e_p/3$ does not matter). The other role is that it characterizes the electromagnetic interaction thus determining α_e (the fine structure constant). This is what makes the electric charge different from mass for there is no "mass quantum". It is natural enough to hypothesize by analogy with the electromagnetic interaction, that a quantum number the "strong charge" might exist. However, the strong interaction is over very short distances, within the confines of a nucleon, and so for it not to manifest itself beyond the immediate confines of a nucleon, the "strong charges" of the quarks making up a nucleon must mutually cancel so that the net "strong charge" of the nucleon is zero. This is the property of "complementarity". Charge invariance (namely the claim that replacement of an electron with charge -e by its anti-particle with charge +e) rules out the possibility that the "strong charge" could have the values +1, -1, and 0 (for replacement of a charged particle by a neutral one is not charge invariant).

Color is the quantum number of "strong charge". It is given the property a-priori that combinations that make up real composite particles are color neutral. Thus, bare color is never observed in the way that the electric charge of the electron can be measured to be e_p. For baryons (nucleon like matter made up of three quarks), color exhibits complementarity in that the different quarks must all have different colors (red, green, and blue) which in combination cancel any net color. For hadrons (mesons made up of a quark-anti-quark pair), (color) charge invariance cancels any net color charge. In this picture, the particles that mediate between quarks, called gluons, possess color and transfer it thus changing the color (but not the kind) of a quark. This is different from the picture of the neutral photon that mediates electromagnetic interactions. This difference reflects itself in the dynamical equations of the two fields. In quantum electrodynamics, the equations are typically linear whereas those in quantum chromodynamics typically are not. This non-linearity is probably from where quark confinement manifests itself.

1.2b Four Forces of Nature (At "Low" Energy)

Currently, there are four unambiguous forces in Nature (at least in this relatively low energy region of the present epoch of the Universe).

Table 1.1

The four forces of Nature

Interaction	Coupling constant		Interaction
	Analytic	Numerical	radius (cm)
Gravity	$Gm^2/\hbar c$	0.6×10^{-38}	infinite
Weak	$g_f m^2 c/\hbar^3$	10^{-5}	10^{-17}
Electro	$e^2/\hbar c$	1/137	infinite
Strong	$a/[\ln (m/m_p)]$ $m >> m_p$	~ 1	10^{-13}

We have seen the importance of the quantum numbers in that they are used to uniquely classify the elementary particles. The spin *s* is particularly important in that all particles fall into either of one of two spin categories. Particles with half-integer spin (1/2, 3/, 5/2) are called fermions and they obey the Pauli exclusion principal which says that no two fermions may occupy the same state. It is precisely this kind of limitation which gives rise to atomic structure. For example, the first period of the periodic system of the elements consists of two elements, H and He. Here the principal quantum number is one and only two possible spin alignments are possible giving rise to the two elements of the period. In successive periods, one generalizes this. Atomic orbits are governed by their angular momentum quantum number L. The electrons in a given orbital or shell (which defines a new period of the elements) all occupy different states with the same orbital energy. This energy degeneracy comes about because of the spin-orbital coupling (s and l) can have more than one combination of states with the same energy. In general, the total number of degenerate states allowed for a given orbital defined by the orbital angular momentum quantum number l is given by:

$$\begin{array}{rcccccc} l = & 0, & 1, & 2, & 3, & 4, & 5,\ldots \\ & s, & p, & d, & f, & g, & h,\ldots \\ 2(2l+1) = & 2, & 6, & 10, & 14, & 18, & 22,\ldots. \end{array} \qquad (1.8)$$

Thus, in the second period, with the orbital angular momentum quantum number being two, there are eight different possible states and thus eight different elements.

This points out the principle characteristic of the fermions, that they constitute the basis of matter. It is the Pauli exclusion principle which prevents the atomic electrons from collapsing to the energetically favorable ground state.

In contrast, particles with whole integer spin (0, 1, 2, ...), called bosons, have no exclusion rule as do the fermions. In principle, one could continue to pack as many as one might choose into a single state. The most elemental bosons are propagators (mediators) of the different forces (fields).

Particles are also classified by their interactions. Strong interacting particles are called *hadrons* (Greek for "heavy"). These particles were originally distinguished by mass from the lighter particles called *leptons* (Greek for "light"). Today, such distinctions in terms of relative mass of the particle are no longer meaningful because there are cases such as the τ lepton which has a mass (m_τ = 1.784 GeV/c^2) or about 1.8 times the mass of the proton (m_p = 0.938 GeV/c^2). The modern distinction between hadrons and leptons is whether or not they participate in strong interactions.

Leptons are those fermions which do not participate in strong interactions and hadrons (both fermions and bosons) do. Hadrons are further subdivided into baryons (fermions with half-integer spin), the lightest of which is the proton, and mesons (bosons with integer spin), the lightest of which is the π-meson ($m_\pi = 0.140$ GeV/c^2).

A table summarizing the elementary particles and the exchange bosons is given below (where S = strong, E = electromagnetic, W = weak, G = gravitational, and n = integer):

Table 1.2

The elementary particles by class

Particle class	Spin	Types of interactions	Mass dependence
Fermions	n/2	independent	independent
Bosons	n	S, E, W, and G	independent
Leptons	n/2	E, W, and G	independent
Mesons	n	S, E, W, and G	independent
Baryons	n/2	S, E, W, and G	$m > m_p$

1.3 The Quest of Unification - Towards a Standard Model

By the 1970's, a Q.E.D. like theory for the weak nuclear force had been developed. The principle difference being massive gauge bosons (now the W^+,W^- and the Z^0) were needed to mediate the short range interaction. The experimental evidence for "partons" existed and Gell-Mann's quark hypothesis was gaining acceptance. Because of the gross similar features of the two separate theories for the electromagnetic and weak forces, a way to a partial unified field theory was now open.

1.3a The Fundamental Particle Model (Leptons, Neutrinos, and Quarks)

The emphasis on the word "particle" is a bit misleading. The wave-particle duality of quantum mechanics gives us another picture besides just the idea that a piece of something collides with another piece of something in an interaction. The field description of interacting waves is just as correct. Theoretical descriptions of particle interaction are based upon "current theories". In these theories, one has two basic kinds of wave-particles involved. The first are the fundamental interacting fermions, point-like particles which are the ultimate (at least as of today in our understanding) constituents of matter. The second are the exchange bosons (also known as the field mediators or propagators) which exchange in an interaction both the external values of energy-momentum and mass and the internal quantum numbers of spin, isospin, charge, and color.

These theories are called current theories because the first one to be developed, quantum electrodynamics (QED) was a quantum mechanical redescription of Maxwell's electromagnetic theory. Classically, one might view any electromagnetic interaction as consisting of two antennas, a transmitting one and a receiving one, both of which form current loops of electrons. Radio waves (photons) exchange the energy from the transmitting antenna to the receiving antenna.

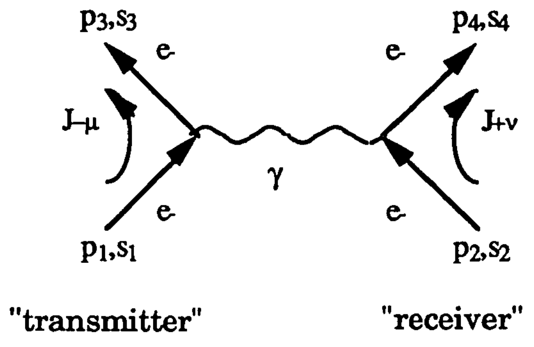


Figure 1.1 Schematic representation of the interaction between two electron current loops mediated by a photon.

Table 1.3

The exchange bosons by interactions

Interaction	Boson	Mass	Spin	Charge	Isospin	Color
Gravity	Graviton	0	2	0	0	No
Weak	W^+, W^-	80	1	+1, -1	1	No
	Z^0	90	1	0	1	No
Electromag	Photon	0	1	0	0	No
Strong	Gluon	0	1	0	0	Yes

Table 1.4

The fundamental fermions by family (flavor)

Electric charge	Flavor	Mass (GeV/c^2)	Flavor	Mass (GeV/c^2)	Flavor	Mass (GeV/c^2)
	leptons (l) spin = 1/2					
0	ν_e	$< 2 \times 10^{-8}$	ν_υ	$< 2.5 \times 10^{-4}$	ν_τ	$< 3.5 \times 10^{-2}$
1	e	5.1×10^{-4}	μ	0.106	τ	1.784
	quarks (q) spin = 1/2					
2/3	u	4×10^{-3}	c	1.5	t	>41
-1/3	d	7×10^{-3}	s	0.15	b	4.7

1.3b Electro-Weak Unification

Weinberg and Salam proposed an electro-weak unification model that introduced a triplet of intermediate bosons (the W^+,W^- and W^0) and a singlet (the B^0) with the neutral bosons mixing so as to form the actually observable bosons:

$$\gamma = \cos\theta\ B^0 - \sin\theta\ W^0$$

$$Z^0 = \sin\theta\ B^0 + \cos\theta\ W^0 \qquad (1.9)$$

where θ is the "Weinberg" angle now known to be given by

$$\sin^2\theta = 0.23 \pm 0.02 \qquad (1.10)$$

and γ is the familiar photon.

At the time, only the first three (flavors, species) of quarks were known, namely the lower massed u, d, and s quarks. One of the problems to unification was the after the discovery of the weak neutral current (making at least part of the weak interaction like the electromagnetic interaction in that charge is not carried by the propagator), only examples in which the strangeness was conserved could be found experimentally. This dilemma was solved by S. Glashow, J. Iliopoulos, and L. Maiani in what is now called the GIM mechanism (after their initials). They proposed the existence of a fourth quark (the c quark, analogous to the u quark), called charm. It was to have a mass close enough to the s quark that the strangeness-changing neutral currents would be exactly canceled out by the charm-changing neutral currents. In 1974, charmonium was discovered in the form of the double named J/Ψ particle.

1.3c Grand Unified Theories (GUTS) and Proton Decay

Many problems remain with the electro-weak unification. First, it does not at all include anything about the strong force (let alone gravity). Second, it requires two unrelated coupling constants. Third, while it presents nicely families (originally two, now three) of quarks, leptons, and neutrinos, it does not really say anything about how these family members are related (most particularly where their masses come from, i.e. we still do not know why m_e has the mass that it does). Finally, all interactions observed so far are of the left-handed or V - A type while the Dirac algebra in no way excludes a-priori interactions of the right-handed or V + A type. Still, it is a premise of the Standard Model that such interactions will be of the more complicated left-handed variety (a combination!) for this fits the data as a sort of maximally parity violating interaction. Further, it is essential that this be put to the test in the third family. The general observation is that as one goes higher in energy things should become more symmetric (early phases in the Universe require less symmetry breaking). Yet, for reasons not understood, nature seems to have taken the path at the threshold of electro-weak interactions to choose this particular broken symmetry. **Why** this should be the case a-priori is still a question. Similarly, along with the observation that the strong force coupling constant should weaken at higher energies while the two electro-weak coupling constants should strengthen, make up motivation for the Grand Unified Theory that hopes to unify the electro-weak and strong interactions.

The major prediction of these theories is that protons decay (diamonds are not forever). An upper bound on the decay rate of protons is easily established when one notes that there are something on the order of 10^{29} protons in the human body. From this one can deduce that the lifetime of a proton must be at least 10^{15} years because if it were any shorter, the resultant ionizing radiation (from 10^{14} protons per year) would kill us and life would not be possible.

Typically, the calculated values are in the range of a lifetime from 10^{29} years to 10^{33} years. One seeks this experimentally by observing over a multiyear period a collection of protons (very pure water) on the order of 10^{32} in number. To date, no protons have yet been observed to spontaneously decay with the IMB experiment reporting out a upper bound of about 4 $x10^{32}$ years.

CHAPTER 2 - The State of the Art - a Review of Previous Theoretical Investigations

2.1 Introduction of a Third Family of Leptons (Quarks) into the Standard Model

At the time of this investigation (1981-1983), with the addition of a third family of leptons and quarks, two questions posed themselves which it was felt could be looked into. The first question concerns itself with universality. Generally speaking, if the idea of higher and higher symmetry appearing in higher energy regimes is correct, then there may be a chance to demonstrate in the third family some greater symmetry than is found in the two lower energy families. A good candidate for this is the experimentally noted reference for electro-weak interactions to be V-A in character rather than say V+A. There is no reason a-priori for this to be the case. Simply, all of the experimental data within the first two families shows only indications of V-A interactions. It is obviously asymmetrical. The second question is one of neutrino masses. It is well known that as one goes up in energy which is to say moves through the families that the associated leptons and quarks (at least in the sense of their identical quark anti-quark pair mesons) increase in mass. Thus, one might call it a theoretical prejudice, it seems reasonable to assume that this pattern replicates itself in the neutrinos. This is all summarized below in Table 2.1:

Table 2.1 The three lepton-quark families

LEPTON GENERATIONS	$\begin{pmatrix} e \\ \nu_e \end{pmatrix}$	$\begin{pmatrix} \mu \\ \nu_\mu \end{pmatrix}$	$\begin{pmatrix} \tau \\ \nu_\tau \end{pmatrix}$
QUARK GENERATIONS	$\begin{pmatrix} u \\ d \end{pmatrix}$	$\begin{pmatrix} s \\ c \end{pmatrix}$	$\begin{pmatrix} t \\ b \end{pmatrix}$

In order to obtain a more detailed understanding of the situation at the time of this investigation, several papers will be reviewed here.

2.2 The Nature of the τ–ν_τ–W Coupling

Bacino, et. al.[1] explore three characteristics of the τ–ν_τ–W coupling. First, they look at the space-time structure of the coupling. Second, they consider the mass of the ν_τ in terms of its upper limit. Finally, they give an estimate on the lower limit of the coupling strength.

The basic idea is that they consider the decay shown in Figure 2.1:

$$\tau^- \rightarrow e^- \overline{\nu}_e \nu_\tau \tag{2.1}$$

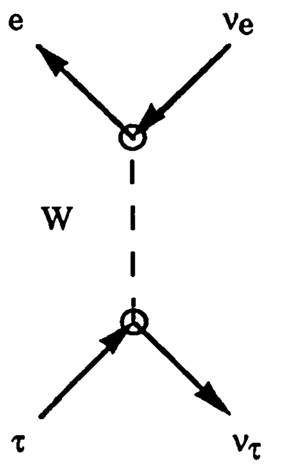


Figure 2.1 The decay of the tauon into and an electron and two neutrinos

For the analysis, a pure V-A coupling was assumed for the e-ν_e vertex. Then, the e momentum spectrum was used to explore the V,A phenomenology, and hence the structure, of the τ– ν_τ vertex.

[1] W. Bacino, et al., Phys. Lett. 42, 749 (1979)

A convenient characterization of the coupling is given by the Michel formula:

$$N(z)dz \approx z^2\left[1 - z + \frac{2}{9}\rho(4z - 3) + r(z)\right] d(z) \qquad (2.2)$$

where

z = electron energy relative to the maximum possible value of E_e / E_e^{max}

ρ = Michel parameter (which characterizes the coupling)

V-A --> $\rho = 0.75$

V+A --> $\rho = 0.0$

which are the only values allowed for a ν_τ mass of zero and pure V or pure A --> $\rho = 0.375$

$r(z)$ = radiative correction term which "softens" the uncorrected spectrum (different for pure V-A and pure V+A)

This is all summarized in Figure 2.2 below:

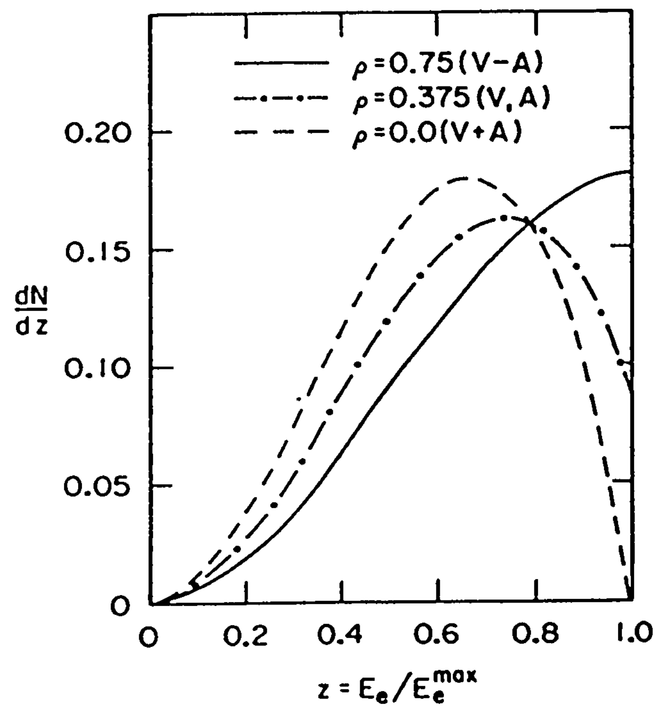


Figure 2.2 Uncorrected (r(z) = 0) electron momentum spectrum in τ rest frame for several values of ρ

The experimental arrangement used to conduct this investigation was the DELCO detector at SPEAR. The center-of-mass energy range was

$$3.57 \text{ GeV} \leq E_{cm} \leq 7.4 \text{ GeV}. \tag{2.3}$$

A number of factors were taken into account in the analysis. First, the electron track was determined from the outer wire spark and the momentum range (below the beam energy and above 0.3 GeV/c to avoid uncertainies in Cherenkov counter detection efficiency). Second was the momentum criterion. Third, account was made for the charmed background (Ψ'' (3770)). Elastic scattering of the e^+e^- events was used to obtain a handle on the measurement and the Coulomb scattering errors. Finally, agreement was made between the Monte Carlo modeling and the experiment.

594 events of the type

$$e^+e^- \rightarrow e^\pm X^\pm \tag{2.4}$$

where X is not e^+ or e^- were found and cuts applied to them. Numerically, the results are expressed in Table 2.2

Table 2.2

Results of the fits to the electron momentum spectrum

Hypothesis	ρ	χ^2	No. of degrees of freedom
V-A	0.75	15.9	17
V+A	0.0	53.7	17
Free fit	0.72 ± 0.10	15.8	16

where the error expressed is purely statistical. Combining their estimate of the systematic errors, their final reported value is a ρ of 0.72 ± 0.15 which is in very good agreement with the V-A hypothesis and nothing else.

Another comparison of the V-A and V+A hypotheses is given in Figure 2.3:

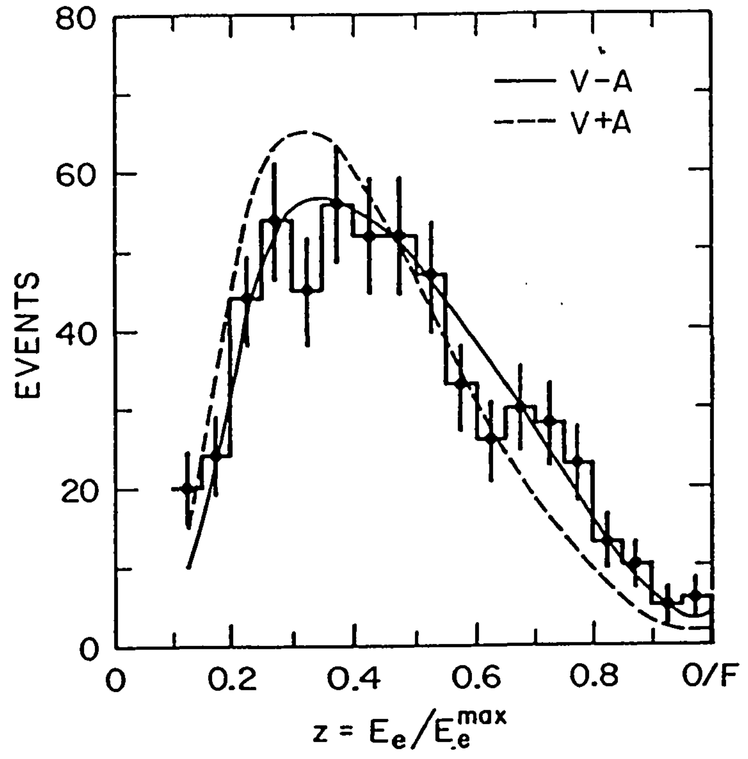


Figure 2.3 The electron momentum spectrum with $z = E_e/E_e^{max}$ in the range 3.57 GeV $\leq E_{cm} \leq$ 7.4 GeV excluding the resonance at Ψ'' (3770). The solid and dashed lines are, respectively, V-A and V+A fits with zero ν_τ mass.

It is possible to make an alternative presentation of the data. This is shown in Figure 2.4.

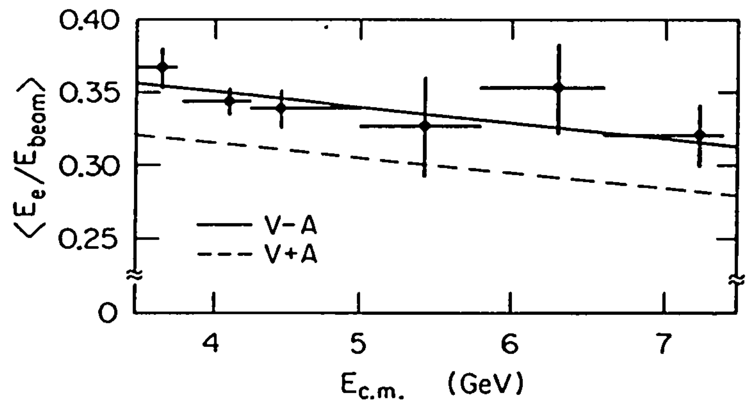


Figure 2.4 The average E_e/E_{beam} for several ranges of center-of-mass energy (indicated by the horizontal error bars). The energy dependences of the predictions of the V-A and V+A hypotheses (indicated respectively, by the solid and dashed lines) result from experimental cuts and measurement errors.

In conclusion, the authors note that the electron momentum spectrum is in good agreement with the V-A hypothesis where

$\rho = 0.75$ theoretical (2.5a)

$\rho = 0.72 \pm 0.15$ experimental[2]. (2.5b)

If one assumes a $\rho = 0.75$, i.e. a pure V-A interaction, then the measured χ^2 variations of ρ give an upper limit for the ν_τ mass[3] of

$$m(\nu_\tau) \leq 250 \text{ GeV} \qquad (2.6)$$

with a 95% confidence level. Additionally, they obtained an upper limit on the τ lifetime of

$$\tau(\tau_0) < 2.3 \times 10^{-12} \text{ sec} \qquad (2.7)$$

again at a 95% confidence level. This implies that the square of the coupling constant at the τ–ν_τ–W vertex is at least 12% (at the 95% confidence level) of the full weak strength[4] (from an assumed $\tau(\tau_0) \sim 2.7 \times 10^{-13}$ sec.).

[2] The currently accepted limit on this is $\rho = 0.73 \pm 0.07$. See the particle data book.

[3] The currently accepted limit on this is $m(\nu_\tau) < 35$ MeV. See the particle data book.

[4] The currently accepted limit on this is $\tau(\tau_0) < (3.04 \pm 0.09) \times 10^{-13}$ sec consistent with 100% full weak strength. See the particle data book.

2.3 Measurement of the τ Lifetime

Feldman et al[5] report a measurement of the τ lifetime. This investigation was carried out with the Mark II detector at the e^+e^- storage ring PEP at SLAC. There, pair production by way of the interaction

$$e^+e^- \dashrightarrow \tau^+\tau^- \tag{2.8}$$

produced τ pairs. Four potential sources of backgound events were identified. These were beam-gas interactions; two photon τ pair production,

$$e^+e^- \dashrightarrow e^+e^- \tau^+\tau^-; \tag{2.9}$$

hadron production,

$$e^+e^- \dashrightarrow \text{hadrons}; \tag{2.10}$$

and radiative Bhabha scattering,

$$\begin{array}{l} e^+e^- \dashrightarrow e^+e^-\gamma \\ \qquad\qquad\quad \lfloor\_\_ \; e^+e^-, \end{array} \tag{2.11}$$

where the photon conversion either was internal or occurred in the 0.09 radiation length of the material between the center of the luminous region and the main drift chamber. A series of cuts were made reducing the data set down to 284 events with 306 three-prong τ decays. Additional cuts were made to reduce the chance that any of the events were scattered or mismeasured resulting in a final sample of 126 three-prong τ decays. Finally, for the remaining data set, the flight distance was calculated between the center of the luminous region and the decay vertex.

[5] G. J. Feldman, et al., Phys. Rev. Lett. 48, 66 (1982)

Making known corrections and combining the systematic and statistical errors in quadrature they report a τ lifetime of

$$\tau_\tau = (4.6 \pm 1.9) \times 10^{-13} \text{ sec} \tag{2.12}$$

Finally, they point out that assuming that the τ couples with the weak current just as strongly as the μ then its lifetime should be given by

$$\tau_\tau = (m_\mu/m_\tau)^5 \tau_\mu B_e = (2.8 \pm 0.2) \times 10^{-13} \text{ sec} \tag{2.13}$$

where B_e is the branching fraction for $\tau \longrightarrow e \nu \bar{\nu}$ (namely 0.176 ± 0.016). Thus, at one standard deviation the τ coupling compared to the weak charged current is 0.66 to 1.02 times the value expected from $\tau-\mu$ universality.

2.4 A Study of the Decay $\tau^- \dashrightarrow \pi^-\nu_\tau$

Blocker et al.[6] investigated the decay τ^- --> $\pi^-\nu_\tau$ determining the branching ratio $B(\tau^- \dashrightarrow \pi^-\nu_\tau)/B(\tau^- \longrightarrow e^-\overline{\nu}_e\nu_\tau)$. They then fitted the p energy spectrum whose end point is determined by the π, τ, and ν_τ masses. From this they determined an upper limit on the ν_τ mass.

The experimental arrangement that they used was the Mark II detector at the e^+e^- storage ring SPEAR at SLAC running with a center-of-mass energy range of $3.52 \leq E_{cm} \leq 6.7$ GeV. The result that they pull out of their data is that experimentally

$$B(\tau^- \dashrightarrow \pi^-\nu_\tau)/B(\tau^- \longrightarrow e^-\overline{\nu}_e\nu_\tau) = 0.66 \pm 0.03 \pm 0.11 \qquad (2.14)$$

which compares favorably with the theoretical value of 0.58. They note that if τ^- --> $\pi^-\nu_\tau$ proceeds via the standard weak axial vector current then this ratio is fixed by known parameters. This is the first time a "correct" measurement was made.

Finally, using only data from the largest block of fixed energy running (5.2 GeV) to minimize systematic variations in efficiencies and background subtraction for data from different center-of-mass energies the π energy spectrum was fitted giving

$$m(\nu_\tau) \leq 250 \text{ MeV} \qquad (2.15)$$

which is the same upper limit reported before by Bacino et al! Figure 2.8 reflects this:

[6] C. A. Blocker, et al., Phys. Lett. **109B**, 119 (1982)

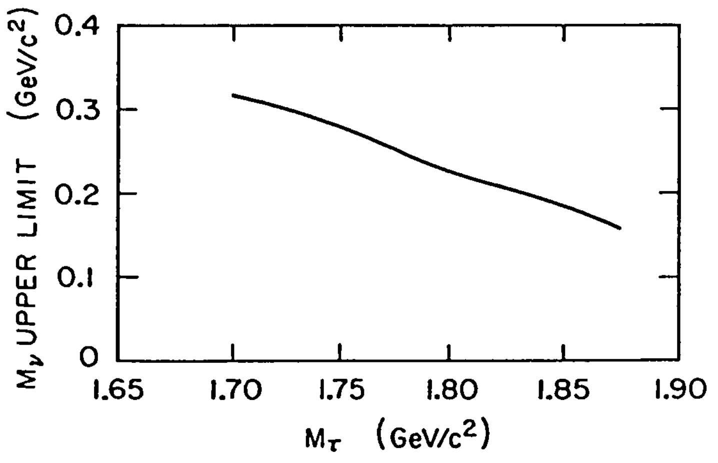


Figure 2.5 Upper limit (95% confidence level) on the mass of the ν_τ as a function of the mass of the τ.

CHAPTER 3 – Decay-product Correlation of $\tau^+ \tau^-$ Production from $e^+ e^-$ Annihilation

3.1 Introduction

The basic idea of this investigation is that an $e^+ e^-$ beam sitting on a center-of-mass energy of 3.57 GeV (double the τ mass[1]) produces a $\tau^+ \tau^-$ pair product. In turn, the τ's decay into products, some of which are observable. In particular, should the products include mesons which are anti-particles of one another then the distribution of momentum and energy is identical in both decays. Such occurrences, although somewhat rare, nevertheless happen in predictable branching ratios. One then carefully measures the correlation angle between the observable mesons. The intention of this calculation is to relate variations in the correlation angle from an exact back to back relationship to phenomena associated with the possibility of a ν_τ mass or higher symmetry in the higher energy coupling of the third generation of leptons.

There are two reasons for carrying out this series of calculations with the τ and its associated ν_τ. First, experimentally[2], at the time of this calculation (1982), the upper bound on the ν_τ mass was found to be

$$m_{\nu_\tau} < 250 \text{ MeV}$$

One can argue for a non-zero ν_τ mass which may be easier to find than other neutrino masses in the sense that it may be heavier. One such arguement notes that as one goes from the first generation of the e to the second generation with the μ to the third generation with the τ, the masses of the leptons increase. While not as directly linked to the neutrinos as are

[1] Particle data book

[2] C.A. Blocker, et. al. 1979

the leptons, nevertheless this same pattern reflects itself in the quarks of the different generations. Now, unlike the photon, there is no strong principle such as the gauge principle to insist upon neutrinos having exactly zero mass. It follows then that most likely neutrinos have a mass but, at least in the case of the ν_e and ν_μ where extensive searching has been underway, this mass is quite small. One should note though the masses of the leptons as shown below in Table 3.1:

Table 3.1

Masses of the Leptons

Generation	Lepton	Mass (MeV)
1	e	0.511
2	μ	105.66
3	τ	1784.2

Thus, one has ratios of the sort

$$m(\mu)/m(e) = 206.77 \text{ and } m(\tau)/m(e) = 3491.6 \qquad (3.1)$$

so that if one were to assume that similar scaling ratios exist for the neutrinos[3], then if the mass of the ν_e is as low as 0.1 eV, this would mean that

$$m(\nu_\mu) = (0.1 \text{ eV}) \times 206.77 = 20.7 \text{ eV}. \qquad (3.2a)$$

[3] an admittedly theoretical prejudice if you like

and

$$m(\nu_\tau) = (0.1\ \text{eV}) \times 3491.6 = 349\ \text{eV} \tag{3.2b}$$

The cosmological consequences of all of this are that should the ν_τ exist in any quantity in the Universe there may be enough mass to close the Universe.

In the early Universe, neutrinos decouple from the expanding big bang before photons because their cross sections of interaction are so much smaller than that of the photons. Assuming only three species of neutrinos, this leads to a background flux that is

3 (neutrino species) x 2 (distinct particle / antiparticle)
x 1 (spin orientations) x 7/8 (fermion statistics) = 21/4

that of the 2.7 ^{0}K background photon number density of approximately 5.5 x 10^2 cm^{-3}. Thus, it is estimated that the total Universe background number density of neutrinos is on the order of 2.9 x 10^3 cm^{-3}. Equipartition leads to 1/3 of all neutrinos (assuming that there are only three lepton generations to deal with) being ν_τ's. Assuming that the most significant contribution to the total neutrino mass was due to the ν_τ's, then one would have an approximate number density[4] of

$$\begin{aligned} n\,(\nu_\tau) &= (21/4) \times (5.5 \times 10^2\ \text{cm}^{-3}) \times (1/3) \\ &= 9.6 \times 10^2\ \text{cm}^{-3} \end{aligned} \tag{3.3}$$

Compare this value with the (critical) energy density n_c to close the Universe

[4] S. Weinberg, 1972

$$n_c = \frac{3H^2_o}{8\pi G}$$

$$= 1.1 \times 10^{-29} \left(\frac{H_o}{75\,\mathrm{km/sec/Mpc}}\right)^2 \mathrm{g/cm^3} \left(\frac{1}{1.78 \times 10^{-33}\ \mathrm{g/eV}}\right)$$

$$= 6.2 \times 10^3\ \mathrm{eV/cm^3}\ \text{for an}\ H_o = 75\ \mathrm{km/sec/Mpc} \qquad (3.4)$$

so that the critical density is met with a minimum ν_τ mass alone of

$$m_{\nu_\tau}(\mathrm{min}) = \frac{n_c}{n(\nu_\tau)} = \frac{6.2 \times 10^3\ \mathrm{eV\,cm^{-3}}}{9.6 \times 10^2\ \mathrm{cm^{-3}}} = 6.5\ \mathrm{eV} << 349\ \mathrm{eV} \quad . \qquad (3.5)$$

It follows that if this string of assumptions has any validity, namely the linear mass scaling of the neutrino masses following the lepton masses, that the m_{ν_e} is small, and that a significant percentage of all neutrinos are ν_τ's, then there is motivation for this calculation.

A second motivation comes from seeking a more sensitive test of the V-A structure of the third generation charged current which is the "signature" of charged weak interactions.

Previous work[5] finds a Michel parameter of

$$\rho = 0.72 \pm 0.15$$

compared with the theoretical value of $\rho = 0.75$ for pure V-A. Thus, this experiment confirms the V-A structure to within $0.15 / 0.75 = 20\%$ of the pure case. Now, the weak interaction is characterized by the (fundamental leptonic) vertex factor

$$\frac{-i\, g_w}{2\sqrt{2}} \gamma^\mu (1 - \gamma^5) \qquad (3.6)$$

5 W. Bacino, et. al. 1978

where the 2's are purely conventional and $g_w = \sqrt{4\pi\alpha_w}$ is the "weak coupling constant". The remaining factor $\gamma^\mu(1 - \gamma^5)$ is important to be understood. If it were just γ^μ alone one would have a purely vector coupling like QED or QCD. Alternatively, the factor $\gamma^\mu\gamma^5$ alone would represent a purely axial vector coupling. Combined, they form a theory that must violate the conservation of parity. In fact, with the vector (V) and axial vector (A) parts having equal weight, this "V-A" theory is a sort of maximum violation of parity. It is an experimental fact that only the combination V-A seems to fit the data for the first and second generations.[6] In other words, a combination such as $\gamma^\mu(1 + \gamma^5)$ is not allowed by the data yet there is no fundamental principle in Physics to exclude this. This is in agreement with the "Standard Model" and its view of the charged current interactions of leptons.

[6] see Chapter 2

3.2 The Problem - Setting up the Kinematics and Geometry

The simplest τ decay[7,8] to consider is the one which results in the π meson which is a pseudo-scalar meson.

$$\tau^+ \rightarrow \pi^+ + \overline{\nu_\tau} \text{ and } \tau^- \rightarrow \pi^- + \nu_\tau$$

The branching ratio for such events is on the order of 10%[9] and thus the occurrence of a $\tau^+ \tau^-$ pair decaying into their respective neutrinos and an observable $\pi^+ \pi^-$ pair is 10% of 10% which is 1% of all such $\tau^+ \tau^-$ events. Experimentally one measures the angular correlation of the $\pi^+ \pi^-$ pair.

The calculation depends upon a number of factors that are calculated independently and then later combined. First, one calculates the spin dependent differential cross section $d\sigma(s_1, s_2)/d\Omega$ of the interaction

$$e^+ e^- \rightarrow \tau^+ \tau^-$$

which is assumed to be completely electromagnetic in character. Second, one calculates the spin dependent τ decay distributions $d\Gamma_1/d\Omega_1$ and $d\Gamma_2/d\Omega_2$ for

$$\tau^+ \rightarrow \pi^+ + \overline{\nu_\tau} \text{ and } \tau^- \rightarrow \pi^- + \nu_\tau$$

Finally, one combines the cross section $d\sigma(s_1, s_2)/d\Omega$ with the decay distributions $d\Gamma_1/d\Omega_1$ and $d\Gamma_2/d\Omega_2$ and integrates with respect to the production angle resulting in a double integral. It will be shown that there is a good single integral approximation to this which lends itself to numerical evaluation as well as a closed form solution to the integral. The

---

[7] G. Alexander et al.

[8] C. A. Blocker et al.

[9] Particle data book

effect of variation in the mass $m(\nu_\tau)$ as well as the degree of variation from pure V-A coupling is then considered.

3.3 Calculation of the Cross Section $e^+ e^- \rightarrow \tau^+ \tau^-$

To carry out the calculation of the cross section, it is first necessary to establish a meaningful frame of reference. The vertex centered frame chosen is shown in Figure 3.1:

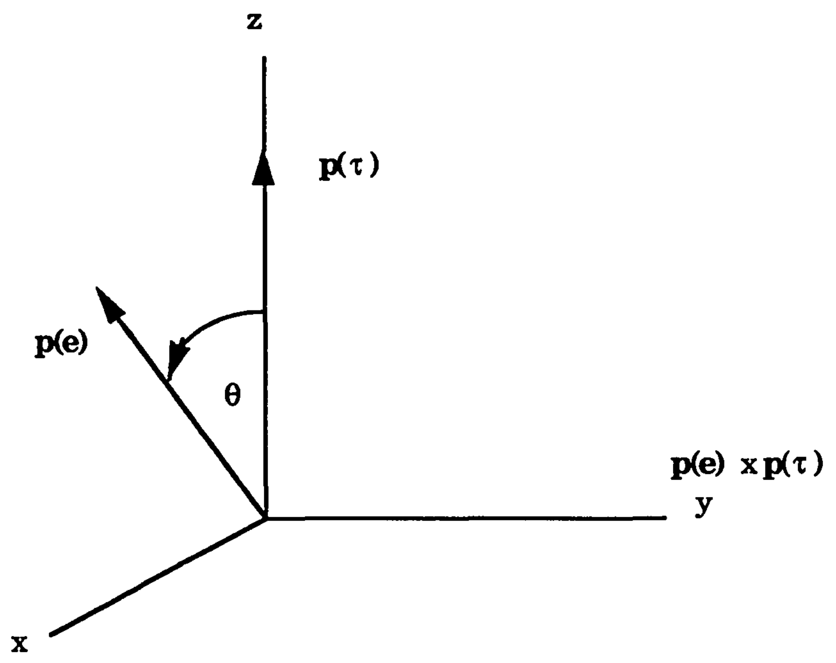


Figure 3.1 The vertex centered frame of reference that is used in the calculation of the cross section.

where the production angle θ is given by

$$\theta = \cos^{-1}\left[\frac{p_{e^-} \cdot p_{\tau^-}}{|p_{e^-}|\,|p_{\tau^-}|}\right] \tag{3.7}$$

and

E = energy of e$^+$(e$^-$) beam

β = velocity of $\tau^+(\tau^-)$

$$= \left[1 - \left(\frac{m_\tau}{E}\right)^2\right]^{\frac{1}{2}} \qquad (3.8)$$

There are two relevant Feynman diagrams for this cross section, namely the direct and the crossed (or annihilation) diagrams. These are shown in Figure 3.2.

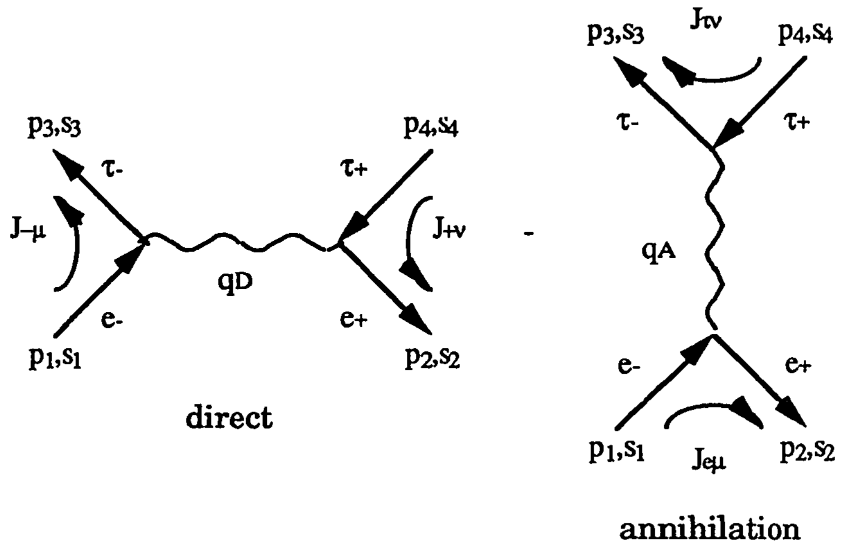


Figure 3.2 The Feynman diagrams that contribute to the cross section of the interaction $e^+ e^- \rightarrow \tau^+ \tau^-$.

Assuming a completely electromagnetic current for the charged lepton interaction, one expresses then, as always, following the order of

adjoint spinor | gamma matrix | spinor

the four relevant currents as

$$J^{-}_{\mu} = \overline{u(3)}\,(i\, g_e\, \gamma_\mu)\, u(1) \tag{3.9a}$$

$$J^{+}_{\nu} = \overline{v(2)}\,(i\, g_e\, \gamma_\nu)\, v(4) \tag{3.9b}$$

$$J^{e}_{\mu} = \overline{v(2)}\,(i\, g_e\, \gamma_\mu)\, u(1) \tag{3.9c}$$

$$J^{\tau}_{\nu} = \overline{u(3)}\,(i\, g_e\, \gamma_\nu)\, v(4) \tag{3.9d}$$

and the photon propagator as

$$P^{\mu\nu}(q) = \frac{-\, i\, g^{\mu\nu}}{q^2} \tag{3.10}$$

The contribution from the first diagram is

$$\begin{aligned} \mathrm{P} &= \int J^{-}_{\mu}\, P^{\mu\nu}(q_D)\, J^{+}_{\nu} \times (2\pi)^4\, \delta(p_1 - p_3 - q) \\ &\quad \times (2\pi)^4\, \delta(p_2 + q - p_4\,)\, \frac{1}{(2\pi)^4}\, d^4q \\ &= \int \left[\overline{u(3)}\,(ig_e\gamma_\mu)\, u(1)\right] \left(\frac{-ig^{\mu\nu}}{g^2}\right) \left[\overline{v(2)}\,(ig_e\gamma_\nu)\, v(4)\right] \\ &\quad \times (2\pi)^4\, \delta(p_1 + p_2 - p_3 - p_4)\, d^4q \\ &= (2\pi)^4\, \frac{ig_e^2}{(p_1 - p_3)^2} \left[\overline{u(3)}\,\gamma^\mu\, u(1)\right]\left[\overline{v(2)}\,\gamma_\mu\, v(4)\right] \\ &\quad \times\, \delta(p_1 + p_2 - p_3 - p_4) \end{aligned} \tag{3.11}$$

so that erasing the residual $(2\pi)^4$ δ function one finds the magnitude for the first (or direct) diagram to be

$$-iM = i\frac{g_e^2}{(p_1 - p_3)^2}\left[\overline{u(3)}\,\gamma^\mu\, u(1)\right]\left[\overline{v(2)}\,\gamma_\mu\, v(4)\right]$$

or

$$M = -\frac{g_e^2}{(p_1 - p_3)^2}\left[\overline{u(3)}\,\gamma^\mu\, u(1)\right]\left[\overline{v(2)}\,\gamma_\mu\, v(4)\right] \tag{3.12}$$

The second diagram represents the annihilation of the electron and positron followed by pair production of the tauons

$$\mathrm{P} = \int J_V^\tau\, P^{\mu\nu}(q_A)\, J_V^e \times (2\pi)^4\, \delta(q - p_3 - p_4) \times (2\pi)^4\, \delta(p_1 + p_2 - q)\frac{1}{(2\pi)^4}\, d^4q \tag{3.13}$$

so that analogous to the first diagram, the amplitude of the second (or annihilation) diagram is

$$M = -\frac{g_e^2}{(p_1 + p_2)^2}\left[\overline{u(3)}\,\gamma^\mu\, v(4)\right]\left[\overline{v(2)}\,\gamma_\mu\, u(1)\right] \tag{3.14}$$

The rule for antisymmetrization says that one includes a minus sign between diagrams that differ only in the interchange of two incoming (or outgoing) leptons, or of an incoming lepton with an outgoing anti-lepton (or vice versa). That is the case here so that the total amplitude is then

$$M = -\frac{g_e^2}{(p_1 - p_3)^2}\left[\overline{u(3)}\,\gamma^\mu\, u(1)\right]\left[\overline{v(2)}\,\gamma_\mu\, v(4)\right] + \frac{g_e^2}{(p_1 + p_2)^2}\left[\overline{u(3)}\,\gamma^\mu\, v(4)\right]\left[\overline{v(2)}\,\gamma_\mu\, u(1)\right] \tag{3.15}$$

In order to find the cross section, one needs to square the amplitude. To do this, one first introduces the projection operators for the electron and positron respectively

$$u(1)\overline{u(1)} = \frac{\overline{\mathbf{P}}_1 + m_e}{2m_e}, \qquad v(2)\overline{v(2)} = \frac{-\overline{\mathbf{P}}_2 + m_e}{2m_e} \tag{3.16}$$

which are spinless (because we do not keep track of initial spins). The notation

$$\overline{\mathbf{P}}_i = \gamma^5 p_i \tag{3.17}$$

indicates the appropriate gamma matrix. The resultant tauon and anti-tauon retain their spins and thus have, respectively, the somewhat more complicated projection operators

$$u(3)\overline{u(3)} = \frac{1 + \gamma_5 \overline{S_3}}{2} \frac{\overline{\mathbf{P}_3} + m_\tau}{2m_\tau},$$

$$v(4)\overline{v(4)} = \frac{1 + \gamma_5 \overline{S_4}}{2} \frac{-\overline{\mathbf{P}_4} + m_\tau}{2m_\tau} \tag{3.18}$$

To find the cross section, one squares the amplitude, and upon expansion introduces the projection operators into the Golden Rule for the Scattering of two bodies in the center-of-momentum frame

$$\frac{d\sigma}{d\Omega} = \left(\frac{\hbar c}{8\pi}\right)^2 \frac{S|M|^2}{(E_1 + E_2)^2} \frac{|\mathbf{p}_f|}{|\mathbf{p}_i|}. \tag{3.19}$$

thus obtaining

$$\begin{aligned}\frac{d\sigma(s_1, s_2)}{d\Omega} = \frac{g_e^4}{(2\pi)^2}\frac{1}{4(p_1 . p_2)}\int \frac{d^3p_4}{2E} \\ \times \int \frac{d^3p_3}{2E}\,\delta^4(p_1 + p_2 - p_3 - p_4) \\ \times \frac{1}{4}\, TR\left[(\overline{p_1} + m_1)\,\gamma_\mu\,(\overline{p_2} - m_2)\,\gamma_\nu\right] \\ \times \frac{1}{4}\, TR\,[(1 + \gamma_5\,\overline{s_3})\,(\overline{p_3} + m_3)\,\gamma_\mu \\ \times\,(1 + \gamma_5\,\overline{s_4})\,(\overline{p_4} - m_4)\,\gamma_\nu]\end{aligned} \tag{3.20}$$

where $m_1 = m_2 = m_e$ and $m_3 = m_4 = m_\tau$. Then, doing the Dirac algebra with $m_e << m_\tau$ one obtains the final spin state dependent cross section:

$$\begin{aligned}\frac{d\sigma(s_1, s_2)}{d\Omega} = \frac{\alpha^2}{16E^2}\,\beta\,[\,(2 - \beta^2\sin^2\theta) \\ +s_{z1}s_{z2}(2\cos^2\theta + \beta^2\sin^2\theta) \\ +s_{x1}s_{x2}(2 - \beta^2)\sin^2\theta - s_{y1}s_{y2}\beta^2\sin^2\theta \\ +(s_{z1}s_{x2} + s_{x1}s_{z2})(1 - \beta^2)^{1/2}\sin 2\theta]\end{aligned} \tag{3.21}$$

3.4 Calculation of the Lifetimes $\tau^- \rightarrow \nu_\tau \pi^-$ and $\tau^+ \rightarrow \overline{\nu_\tau} \pi^+$

The next step in the calculation is to find the decay rates of the decaying tauons. Similar to before, the frame of reference for the calculation is the center-of-momentum frame. In this case, one considers the decaying tauon to be "stationary" and thus the resultant pion and neutrino equally share the momentum. Assuming the decay is mediated by the $W^\pm$ the relevant Feynman diagram is given in Figure 3.3:

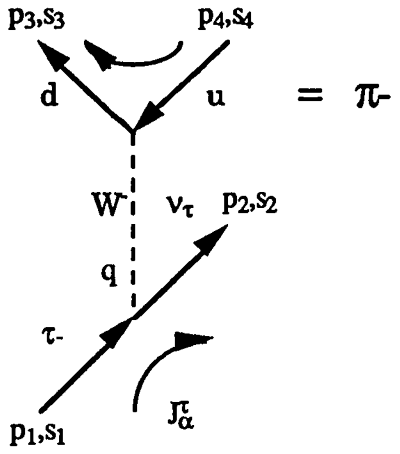


Figure 3.3 The Feynman diagram which represents the decay $\tau^- \rightarrow \nu_\tau \pi^-$.

Here the momentum equation, with P_π representing the momentum of the composite pion, is

$$P_1 = P_\pi + P_2 \quad | \quad \mathbf{P}_1 = 0 \tag{3.22}$$

and the amplitude of the "decay" is given by

$$- i M_{fi} = \langle f | J^{\tau}_{\alpha} P^{\alpha\beta}(q) J^{\pi}_{\beta} | i \rangle \tag{3.23}$$

where the relevant propagator is given by

$$P^{\alpha\beta}(q) = \frac{- i g^{\alpha\beta}}{q^2} = \frac{- i g^{\alpha\beta} \cos\theta_c}{2 M_w^2} \tag{3.24}$$

with θ_c being the Cabbibo angle. One now performs a calculation in three parts. The first part is the spin independent (i.e. isotropic only) part. For a pure V-A (i.e. left handed) case one has the following weak currents:

$$J^{\pi}_{\beta} = \overline{d}\left(g\,\gamma_\beta\left(\frac{1 - \gamma_5}{2}\right)\right) u \tag{3.25a}$$

where the bare d and u stand for the down and up quarks (particles 3 and 4 in the diagram) making up the observable pion, and

$$J^{\tau}_{\alpha} = \overline{u(1)}\left(g\,\gamma_\alpha\left(\frac{1 - \gamma_5}{2}\right)\right) u(2) \tag{3.25b}$$

where $u(1)$ and $u(2)$ represent the tauon and tau neutrino particles respectively. Squaring M_{fi} one obtains

$$\langle |M_{fi}|^2 \rangle = \frac{1}{2}\left(\frac{G_F}{\sqrt{2}} \cos\theta_c f_\pi\right)^2$$
$$\times \sum_{\text{polar}} \left[u(2)\overline{u(2)}\, \overline{p_\pi}\, (1 - \gamma_5) u(1) \overline{u(1)}\, (1 - \gamma_5) \overline{p_\pi}\right] \tag{3.26}$$

Letting $m_1 = m_\tau$ and $m_2 = m_{\nu_\tau}$, this is evaluated using the spinless projection operators as before (both $u(1)$ and $u(2)$ are particles) only in this case one sums over all polarizations. Performing the Dirac algebra, one finds

$$\langle |M_{fi}|^2 \rangle = \left(G_F \cos\theta_c f_\pi\right)^2 \frac{1}{4m_\nu m_\tau} \times [(m^2{}_\tau - m^2{}_\nu)^2 - m^2{}_\pi (m^2{}_\nu + m^2{}_\tau)] \tag{3.27}$$

which is a constant and so the decay is isotropic!

Further, this is physically reasonable because when $m_\nu \to 0$

$$[(m^2{}_\tau - m^2{}_\nu)^2 - m^2{}_\pi (m^2{}_\nu + m^2{}_\tau)^2] = [m^2{}_\tau (m^2{}_\tau - m^2{}_\pi)] \tag{3.28}$$

which means that this process is an allowed decay since $m_\tau > m_\pi$. The final massless ν_τ must be oppositely directed to the final massive π^-. Similarly, in order to conserve angular momentum, the angular momentum axial vectors must be oppositely directed as shown in Figure 3.4.

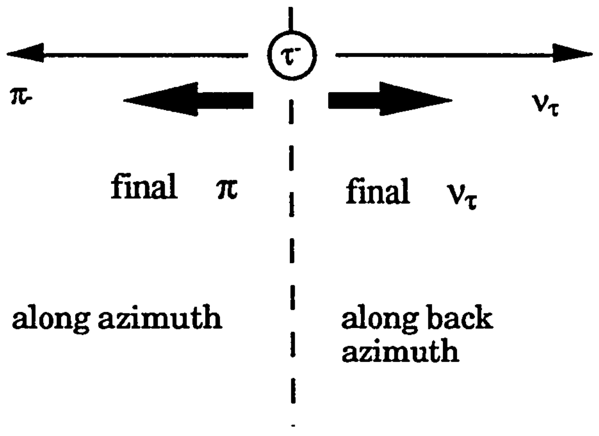


Figure 3.4 The angular momentum axial vectors that are representative of the decay products.

Using $\langle |M_{fi}|^2 \rangle$ one then has for the decay

$$\frac{d\Gamma_1}{d\Omega_1} = \frac{\left(G_F f_\pi \cos\theta_c\right)^2}{64\,\pi^2} \sqrt{\left(1 + \frac{m_\pi}{m_\tau}\right)^2 - \frac{m_\nu^2}{m_\tau^2}}$$
$$\times \sqrt{\left(1 - \frac{m_\pi}{m_\tau}\right)^2 - \frac{m_\nu^2}{m_\tau^2}} \left[\left(m_\tau^2 - m_\nu^2\right)^2 - m_\pi^2\left(m_\tau^2 + m_\nu^2\right)\right] \qquad (3.29)$$

Now, using the $m_\nu = 0$ case as a guide, one defines (the isotropic) A as

$$\left(m_\tau^2 - m_\nu^2\right)^2 - m_\pi^2\left(m_\tau^2 + m_\nu^2\right) = m_\tau^2\left(m_\tau^2 - m_\pi^2\right) A \qquad (3.30)$$

then

$$\frac{d\Gamma_1}{d\Omega_1} = \frac{\left(G_F f_\pi \cos\theta_c\right)^2}{64\,\pi^2} \sqrt{\left(1 + \frac{m_\pi}{m_\tau}\right)^2 - \frac{m_\nu^2}{m_\tau^2}}$$
$$\times \sqrt{\left(1 - \frac{m_\pi}{m_\tau}\right)^2 - \frac{m_\nu^2}{m_\tau^2}} \left[m_\tau^2\left(m_\tau^2 - m_\pi^2\right) A\right] \qquad (3.31)$$

The second part of this calculation is the spin dependent (i.e. isotropic plus anisotropic) part of the pure V-A (i.e. left-handed) case. Here the currents and the expression for $\langle|M_{fi}|^2\rangle$ are identical but the decaying τ is considered polarized thus one has the projection operators

$$\sum_{\text{polar}} u(1)\overline{u(1)} = \frac{1 + \gamma_5 \overline{s_1}}{2} \frac{\overline{p_1} + m_\tau}{2m_\tau} \qquad (3.32a)$$

which has spin (the "1" indicating the τ particle) and

$$\sum_{\text{polar}} u(2)\overline{u(2)} = \frac{\overline{p_2} + m_\nu}{2m_\nu} \qquad (3.32b)$$

which has no spin (the "2" indicating the ν_τ particle) since the experiment is insensitive to the spin orientation of the ν_τ but ultimately is sensitive to the spin orientation of the τ with it decaying into a detectable π. Doing the Dirac algebra, one obtains terms of the form

$$\begin{aligned}(2P_\pi \,.\, P_\nu)\,(P_\pi \,.\, P_\tau) - m_\pi^2\,(P_\nu \,.\, P_\tau) \\ &= \frac{1}{2}\left[(m_\tau^2 - m_\nu^2)^2 - m_\pi^2\,(m_\tau^2 + m_\nu^2)\right] \\ &= \frac{1}{2} m_\tau^2\,(m_\tau^2 - m_\pi^2)\,A\end{aligned} \tag{3.33}$$

which is *isotropic* and $P_\pi \,.\, P_\tau$ is a product of the appropriate four-vectors, etc. and

$$\begin{aligned}-(2P_\pi \,.\, P_\nu + m_\nu^2)\,m_\tau\,(S \,.\, P_\pi) \\ &= \frac{1}{2}(m_\tau^2 - m_\nu^2)\sqrt{(m_\tau + m_\pi)^2 - m_\nu^2} \\ &\quad\times \sqrt{(m_\tau - m_\pi)^2 - m_\nu^2}\;\mathbf{S} \,.\, \mathbf{P}_\pi \\ &\equiv \frac{1}{2} m_\tau^2\,(m_\tau^2 - m_\pi^2)\,B\;\mathbf{S} \,.\, \mathbf{P}_\pi\end{aligned} \tag{3.34}$$

which is *anisotropic* and $\mathbf{S} \,.\, \mathbf{P}_\tau$ is a product of two three-vectors. Note that as $m_\nu = 0 \rightarrow A = B$. Additionally, for convenience one defines the ratios

$$a = \frac{m_\pi}{m_\tau} \quad \text{and} \quad \eta = \frac{m_\nu}{m_\tau} \tag{3.35}$$

then

$$\begin{aligned}\frac{d\Gamma_1}{d\Omega_1} &= \frac{(G_F f_\pi \cos\theta_c)^2}{64\,\pi^2}\sqrt{(1+a)^2 - \eta^2} \\ &\quad\times \sqrt{(1-a)^2 - \eta^2}\left[m_\tau^2\,(m_\tau^2 - m_\pi^2)\,(A + B\,\mathbf{S}.\mathbf{P}_\pi)\right]\end{aligned} \tag{3.36}$$

i.e. an expression containing an isotropic part A plus an anisotropic part B.

This brings us to the third part of the calculation.

Now, one wishes to extend this calculation to one which is mixed in terms of both V-A and V+A contributions. As in the second part, this is spin dependent being a combination of both isotropic and anisotropic parts. One starts out by generalizing the currents such that

pure V-A		mixed V-A and V+A
$(1-\gamma_5)$	$\rightarrow$	$(1-\gamma_5)+\varepsilon(1+\gamma_5)$

(3.37)

then the appropriate weak currents are

$$J^{\pi}_{\beta} = \bar{d}\left(\frac{g}{2}\gamma_{\beta}\left((1-\gamma_5)+\varepsilon(1+\gamma_5)\right)\right)u \tag{3.38a}$$

and

$$J^{\tau}_{\alpha} = \overline{u(1)}\left(\frac{g}{2}\gamma_{\alpha}\left((1-\gamma_5)+\varepsilon(1+\gamma_5)\right)\right)u(2) \tag{3.38b}$$

(again where "1" indicates the τ particle and "2" indicates the ν_τ particle).

The expression for $\langle|M_{fi}|^2\rangle$ in terms of J^{π}_{β} and J^{τ}_{α} is the same as before with the projection operators being those used in the spin dependent calculation of before. The Dirac algebra leads to the terms

$$\begin{aligned}(1+\varepsilon^2)\left[\left(m_\tau^2-m_\nu^2\right)^2-m_\pi^2\left(m_\tau^2+m_\nu^2\right)\right]+4\,\varepsilon\; m_\nu\, m_\tau\, m_\pi^2 \\ \equiv m_\tau^2\left(m_\tau^2-m_\pi^2\right)A\end{aligned} \tag{3.39}$$

for the *isotropic* component and

$$
\begin{aligned}
&(1-\varepsilon^2)\left(m_\tau^2 - m_\nu^2\right)\sqrt{(m_\tau + m_\pi)^2 - m_\nu^2} \\
&\qquad \times \sqrt{(m_\tau - m_\pi)^2 - m_\nu^2}\ \mathbf{S} \,.\, \mathbf{P}_\pi \\
&\qquad \equiv m_\tau^2\left(m_\tau^2 - m_\pi^2\right) B\ \mathbf{S} \,.\, \mathbf{P}_\pi
\end{aligned}
\tag{3.40}
$$

for the *anisotropic* component. These are now the generalized expressions for A and B which can now be used in the expression for $d\Gamma_1 / d\Omega_1$ given above.

Finally, an analogous expression for $d\Gamma_2 / d\Omega_2$ results with the replacement $B \rightarrow -B$.

3.5 Combination and Integration

Now, one wishes to combine the spin dependent production cross section $d\sigma(s_1,s_2) / d\Omega$ with the spin dependent decay widths $d\Gamma_1 / d\Omega_1$ and $d\Gamma_2 / d\Omega_2$ to find the overall correlation angle dependent cross section $d\sigma(\theta_{\mathrm{corr}}) / d\Omega$ for producing back to back observable pions. This is then integrated with respect to the correlation angle θ_{corr}. The first step is to perform the combination. From above (symbolically)

$$\frac{d\Gamma_1}{d\Omega_1} \approx (A + B\,\mathbf{S}\,.\,\mathbf{P}_\pi)$$
$$\rightarrow \Gamma_1 \approx \int A\, d\Omega_1 \approx A \tag{3.41}$$

and

$$\frac{d\Gamma_2}{d\Omega_2} \approx (A - B\,\mathbf{S}\,.\,\mathbf{P}_\pi)$$
$$\rightarrow \Gamma_2 \approx \int A\, d\Omega_2 \approx A \tag{3.42}$$

The spin dependent components are combined by casting them as probabilities and taking the product, namely

$$\frac{d\sigma(\theta_{\mathrm{corr}})}{d\Omega} = \frac{d\sigma(s_1,s_2)}{d\Omega}\left[\frac{1}{\Gamma_1}\frac{d\Gamma_1}{d\Omega_1}\right]\left[\frac{1}{\Gamma_2}\frac{d\Gamma_2}{d\Omega_2}\right] \tag{3.43}$$

substituting in the above

$$\frac{d\sigma(\theta_{\mathrm{corr}})}{d\Omega} = \frac{d\sigma(s_1,s_2)}{d\Omega}\frac{1}{A}\int (A + (B\,\mathbf{S}.\mathbf{P}_\pi))d\Omega_1$$
$$\times \frac{1}{A}\int (A - (B\,\mathbf{S}.\mathbf{P}_\pi))d\Omega_2 \tag{3.44}$$

then

$$\frac{d\sigma(\theta_{\mathrm{corr}})}{d\Omega} = \frac{d\sigma(s_1,s_2)}{d\Omega} \int d\Omega_1\, d\Omega_2 \left[1 - \left(\frac{B^2}{A^2}\right)\right] \tag{3.45}$$

Having combined the pieces, one needs to carry out the integration. With

$$d\Omega = d\cos\theta\, d\phi \tag{3.46}$$

then integrating out $d\Omega$ and introducing the corrolation angle θ_{corr}

$$z = \cos\theta_{\mathrm{corr}} = \frac{\mathbf{p}_1 \cdot \mathbf{p}_2}{|\mathbf{p}_1|\,|\mathbf{p}_2|} \tag{3.47}$$

with

$$p_{1x} = p_1 \sin\theta_1 \cos\phi_1 \tag{3.48a}$$

$$p_{2y} = p_1 \sin\theta_1 \sin\phi_1 \tag{3.48b}$$

$$p_{1z} = p_1 \cos\theta_1 \tag{3.48c}$$

etc. and letting

$$x = \cos\theta_1 \quad \text{and} \quad y = \cos\theta_2 \tag{3.49}$$

one obtains in particular (and dropping the explicit θ_{corr} reference)

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{16\,\pi^2} \int dx\ dy\ d\phi_1\, d\phi_2\, \delta[u + v\cos\left(\phi_1 - \phi_2\right) - z] \tag{3.50a}$$

$$\times\left[1 - \frac{1}{3-\beta^2}\left(\frac{B}{A}\right)^2 \left\{x\,y\ (1+\beta^2) + (1-x^2)^{1/2}\ (1-y^2)^{1/2}\ C\right\}\right]$$

such that

$$C = (1 - \beta^2)\cos\left(\phi_1 - \phi_2\right) + \cos\left(\phi_1 + \phi_2\right) \tag{3.50b}$$

and with u and v being complicated functions of the form

$$u = u(x, y, \beta, a, \eta)$$

$$v = v(x, y, \beta, a, \eta) \tag{3.51}$$

fixing x and y and using the δ function to integrate out ϕ_1 and ϕ_2 results in

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{4\pi}\int_{-1}^{1} dx \int_{y_1}^{y_2} dy \left[v^2 - (u - z)^2\right]^{-1/2} \times \left\{1 - \frac{1}{3 - \beta^2}\left(\frac{B}{A}\right)^2 F\right\} \tag{3.52a}$$

such that

$$F = W_1\ W_2\ z + \beta'^2 + \beta' (x - y) + \beta^2\ x\, y \tag{3.52b}$$

where

$$y_1, y_2 \equiv \text{roots of } f(y) = v^2 - (u - z)^2 = 0 = g(x,z) \tag{3.52c}$$

and

$$W_1 = W_1(x, \beta, a, \eta) \tag{3.52d}$$

$$W_2 = W_2(y, \beta, a, \eta) \tag{3.52e}$$

$$\beta' = \beta'(\beta, a, \eta) \tag{3.52f}$$

This is the *exact* expression which is a double integral.

3.6 Single Integral Approximation

In the limit $\beta \rightarrow 0$, the double integration can be done exactly yielding

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{2}\left[1 - \frac{1}{3}\left(\frac{B}{A}\right)^2 z\right] \tag{3.53}$$

which shows that the cross section is greatest for $\theta_{corr} = \pi$ (i.e. $z = -1$). In particular, the largest effect occurs for the largest value of

$$\left(\frac{B}{A}\right)^2 = 1 \tag{3.54}$$

which holds at $\varepsilon = \eta = 0$ (i.e. pure V-A [$\varepsilon = 0$] and $m_\nu = 0$ [$\eta = 0$]). This is reflected in Figure 3.5:

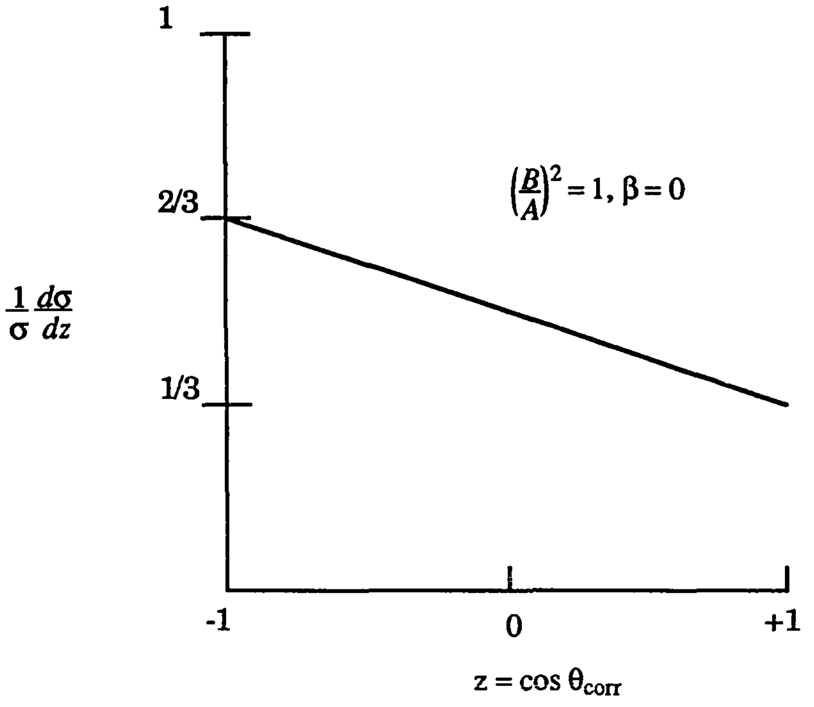


Figure 3.5 Plot showing the variation in cross section with the cosine of the corrolation angle.

and for $\varepsilon = 0$ with $\eta << 1$

$$\frac{B}{A} = 1 - \frac{2a^4}{(1-a^2)^2}\eta^2 \qquad (3.55)$$

which means it is not very detectable since

$$a = \frac{m_\pi}{m_\tau} = 0.08 \qquad (3.56)$$

and

$$\eta = \frac{m_\nu}{m_\tau} < 0.14 \qquad (3.57)$$

is known by experiment. However, some information can still be obtained if $d\sigma / dz$ is measured and limits on $(B/A)^2$ determined. A plot of constant $(B/A)^2$ contours from which the allowed region in ε and η gives:

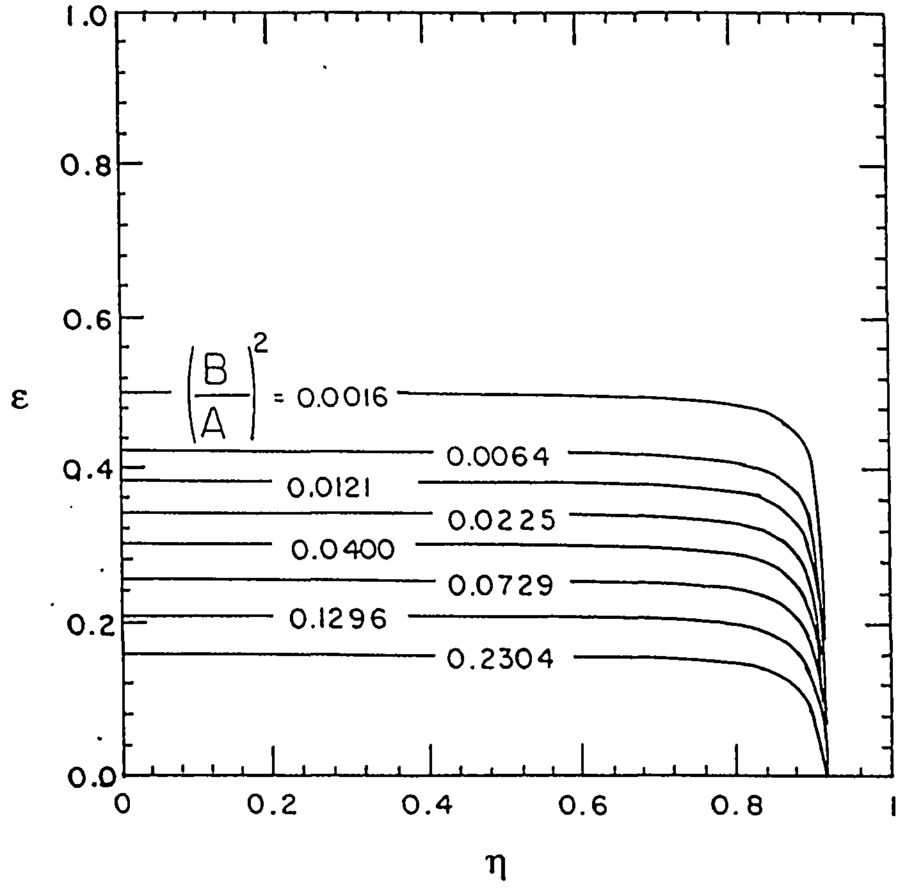


Figure 3.6 Plot which displays a constant contour of $\eta = (B\ / A)^2$ vs. ε.

As β goes from 0 to 1, the cross section is even further enhanced around $z\ = -1$ by the Lorentz boosts given to the τ^+ and τ^- which as shown above even at b = 0 prefer to be oppositely directed. The β = 0 case suggests an approximation:

$$v = \left(1 - x^2\right)^{1/2} \left(1 - y^2\right)^{1/2} \tag{3.58a}$$

$$u = x\,y \tag{3.58b}$$

then for $f(y) = v^2 - (u - z)^2 = 0$

$$v^2 = (z - u)^2 \tag{3.59}$$

$$\left(1 - x^2\right)\left(1 - y^2\right) = (z - x\,y)^2$$

or

$$y^2 - (2\,x\,z)\,y + (z^2 + x^2 - 1) = 0 \tag{3.60}$$

then one finds the roots

$$y_1 = x\,z\; - \left(1 - x^2\right)^{1/2}\left(1 - z^2\right)^{1/2} \tag{3.61a}$$

$$y_2 = x\,z\; + \left(1 - x^2\right)^{1/2}\left(1 - z^2\right)^{1/2}\;,\; y_1 \le y_2 \tag{3.61b}$$

then

$$\frac{y_2 - y_1}{2} = \left(1 - x^2\right)^{1/2}\left(1 - z^2\right)^{1/2} \tag{3.62}$$

$$y_0 = \frac{y_2 + y_1}{2} = x\,z \tag{3.63}$$

evaluating $f(y)$ at $y\; = y_0$

$$\begin{aligned} f(y_0) &= v^2 = (z - u)^2 \\ &= \left(1 - x^2\right)^{1/2}\left(1 - z^2\right)^{1/2} - (z - x\,y_0)^{1/2} \\ &= \left(1 - x^2\right)\left(1 - z^2\right) \end{aligned} \tag{3.64}$$

so that (at $y\; = y_0$)

$$\left(\frac{y_2 - y_1}{2}\right)\frac{1}{\left[v^2 - (z - u)^2\right]^{1/2}} = 1 \tag{3.65}$$

and in lieu of the double integral one has the single integral approximation

$$\frac{1}{\sigma}\frac{d\sigma}{dz} \approx \frac{1}{4}\int_{-1}^{1} dx \left(\frac{y_2 - y_1}{2}\right) \frac{1}{\left[v^2 - (z-u)^2\right]^{1/2}} \times \left\{1 - \frac{1}{3-\beta^2}\left(\frac{B}{A}\right)^2 F\right\} \text{ at } y = y_0 = \frac{y_1 + y_2}{2} \tag{3.66}$$

Figures 3.7 and 3.8 are numerical evaluations of this integral for different values of β and $(B / A)^2$ respectively.

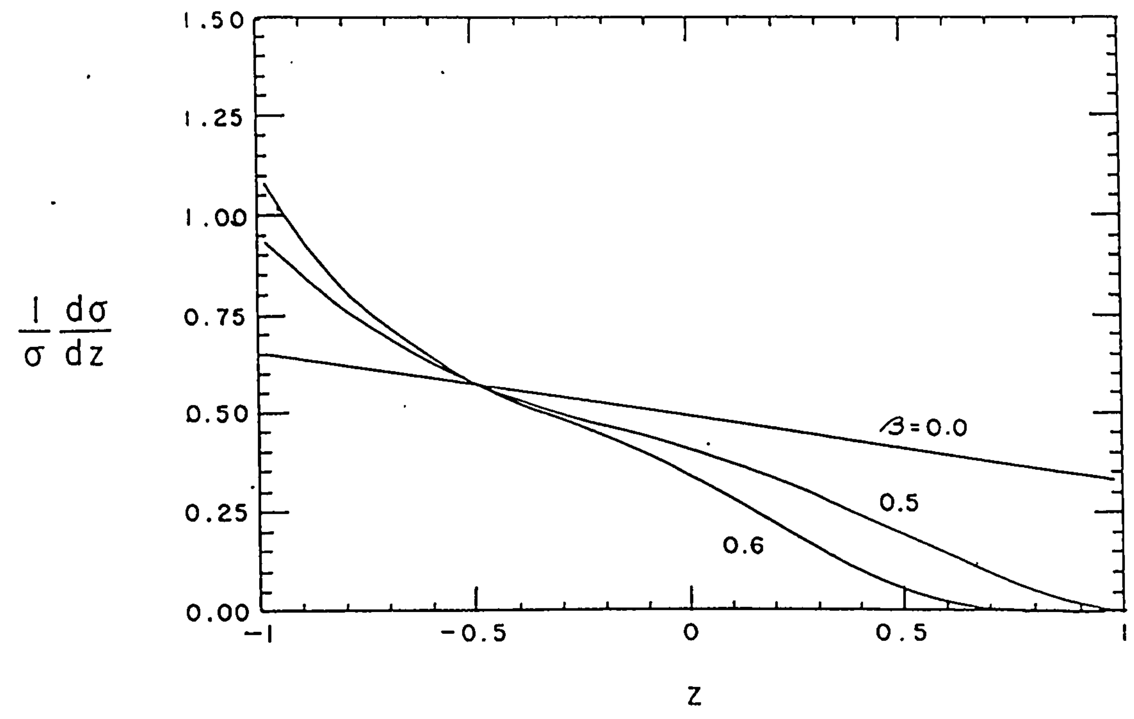


Figure 3.7 This plot shows the cosine of the correlation angle of the observable pions vs. the differential cross section of the interaction with respect to different values of β.

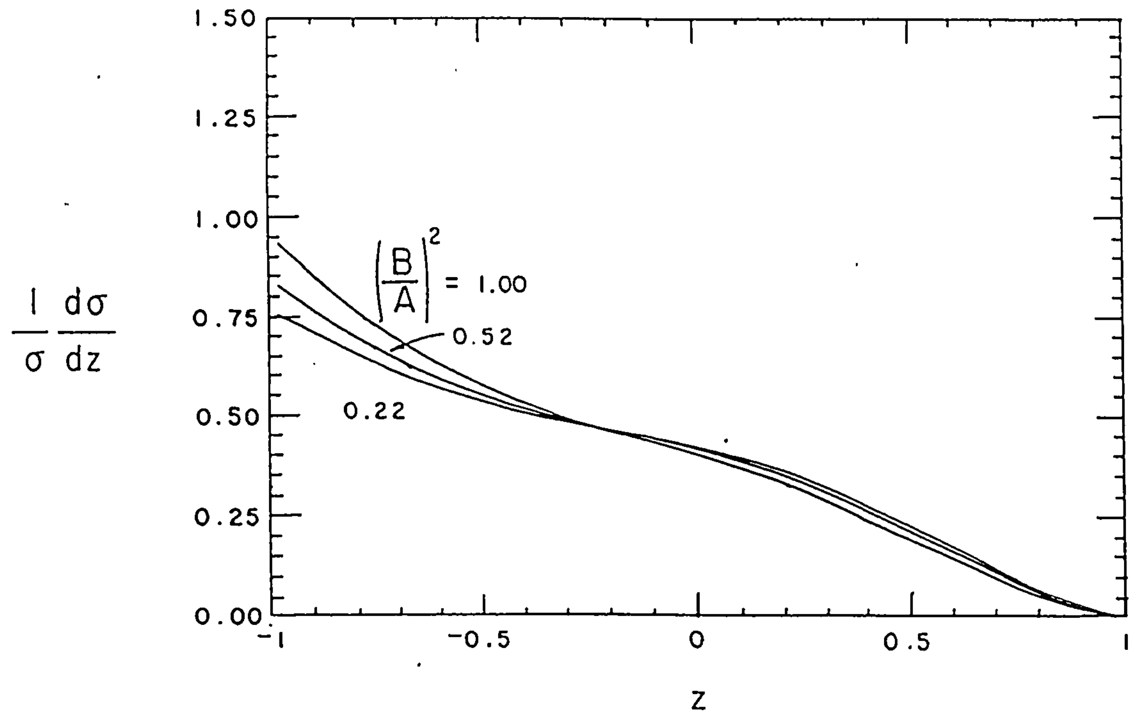


Figure 3.8 This plot shows the cosine of the correlation angle of the observable pions vs. the differential cross section of the interaction with respect to different values of $(B / A)^2$.

3.7 Closed Form Solution

Alternatively, if one is to define

$$z' \equiv x\,y + \left(1 - x^2\right)^{1/2} \left(1 - y^2\right)^{1/2} \cos\left(\phi_1 - \phi_2\right) \tag{3.67}$$

then the integration can be done exactly obtaining a relationship identical in form to (3.53) which now holds for all values of β.

$$\frac{1}{\sigma}\frac{d\sigma}{dz'} = \frac{1}{2}\left[1 - \frac{1}{3}\left(\frac{B}{A}\right)^2 z'\right] \tag{3.68}$$

3.8 Conclusions on Observable Phenomenology (Especially with Respect to Generation Universality)

A - The question arises as to how z' can be measured in the laboratory. The required angles ϕ_1, ϕ_2, θ_1 and θ_2 are all defined in terms of the $\tau^+\tau^-$ axis. Thus, the task is to develop a procedure by which this axis can be unambiguously identified. Such a procedure is to let the measured $\pi^+\pi^-$ momenta be

$$\mathbf{p}_1 = \mathbf{p}_{\pi^+} = \left(p_{1x}, 0, p_{1z}\right) \tag{3.69a}$$

and

$$\mathbf{p}_2 = \mathbf{p}_{\pi^-} = \left(0, 0, p_{2z}\right) \tag{3.69b}$$

then the $\tau^+\tau^-$ axis will be the new z axis obtained by two rotations namely through the angle ϕ about z and then the angle θ about about the new x axis. The third Euler rotation accomplishes nothing since it would not change the direction of the new z axis. Thus, in terms of the new coordinate system one has

$$\begin{aligned}\mathbf{p}_1 = (&p_{1x}\cos\phi, -p_{1x}\cos\theta\sin\phi + p_{1z}\sin\theta,\\ &p_{1x}\sin\theta\sin\phi + p_{1z}\cos\theta)\end{aligned} \tag{3.70a}$$

and

$$\mathbf{p}_2 = (0, p_{2z}\sin\theta, p_{2z}\cos\theta) \tag{3.70b}$$

From this point one boosts $\mathbf{p}_1$ and $\mathbf{p}_2$ back onto the τ^+ and τ^- rest frames respectively. Now, since $\tau \rightarrow \pi\nu$ is a two-body decay, the momentum of the π in the τ rest frame is fixed. Thus, there will be two equations, one for each π momentum in terms of ϕ and θ. Finally, one solves for ϕ and θ and consequently obtains the $\tau^+\tau^-$ axis.

B - Assuming the reported experimental results, the cases of $m_\nu = 0$ and $m_\nu = 250$ MeV are indistinguishable by this approach.

C - The effects on the cross section are roughly opposite in character for increasing m_ν to that of increasing the amount of V+A current in a predominant V-A current. Thus, they mask each other allowing the possibility of a finite $m_\nu \neq 0$ with some V+A coupling. However, the result of conclusion B above means that this experiment may be more sensitive to looking for some V+A current component than in determining an m_ν.

PART II -- EXPERIMENTAL CONSIDERATIONS

CHAPTER 4 – Introduction to the Macrocosmos

4.1 The Big -- Cosmology

If Astronomy is man's oldest observational science, then Cosmology, or at least speculation as to our origins must be man's oldest theoretical adventure. Certainly, the subject is sufficiently important to the species to be at the very core of man's collective religious experience from the very beginning of recorded history if not earlier. In the modern context, Cosmology is the subject of things "big". By big one means literally the size of the Universe.

If one theme characterizes developments in physics in the 1980's, it is probably the union of the studies of the very small, particle physics, and the very big, Cosmology. 20th century cosmology has been driven by the discovery, both theoretically, by Einstein in the General Theory of Relativity (1915 - although not correctly interpreted at the time), and observationally, by E. P. Hubble (1929), that the Universe is expanding. Later work (in the 1940's and 1950's) was devoted to the questions of nucleosynthesis. From this evolved an understanding that only the lightest of the elements could have been made in some great Cosmic beginning and that other elements could only have been made later in such energetically favorable phenomena as stellar evolution and supernova.

In the early 1960's, A. Penzias and R. W. Wilson tried to calibrate a horn antenna with a maser for the receiving amplifier. Such an arrangement should be essentially noise free yet they persistently found an isotropic noise at 7 cm whose intensity was independent of such things as night or day or time of year. This was equivalent to a blackbody noise source whose characteristic temperature is about 3 ^oK. At just about the same time, P. J. E. Peebles had carried out calculations which showed that if indeed the Universe had begun as a sort of singularity in a Big Bang then there should be a fossil radiation of just the characteristics measured.

The understanding now that the Universe began as a collection of all the matter-energy of the Universe in a singularity with energies far beyond the reach of particle accelerators, was a natural to bring particle physics into the picture. Here were the energies to unify all force fields. The current theories used to describe the early Universe are indeed the highest energy theories of all! Thus, the connection between Cosmology and particle physics is made. The seemingly two separate disciplines, the study of the largest scale in the Universe and the study of the smallest scale in the Universe completely fed on one another in the 1980's. The current Cosmological theories are known as "inflationary Big Bang" theories which encompass the idea of an originating singularity which "explodes" (the Big Bang) resulting in an expanding Universe. The inflationary part is a purely high energy physics introduction which through the application of phase transistions has been able to get rid of some earlier problems (particularly the so-called "horizon" problem). In general, cosmological observations can be used to constrain particle theories and particle observations can be used to constrain Cosmological theories. For example, even earlier than the time when no longer every photon was automatically reabsorbed ("Let there be light!") which today we see as the 3 ^0K background, the neutrinos should have been decoupled from the early Universe. Arguments along these lines lead to conclusions that no more than four species of neutrinos are allowed (and thus four families or leptons and quarks). Experimental verification of this at CERN in Europe and at SLAC in the United States of America through the availability to produce large quantities of the Z^0 is one of the "hottest" research efforts under way at the time of this writing.

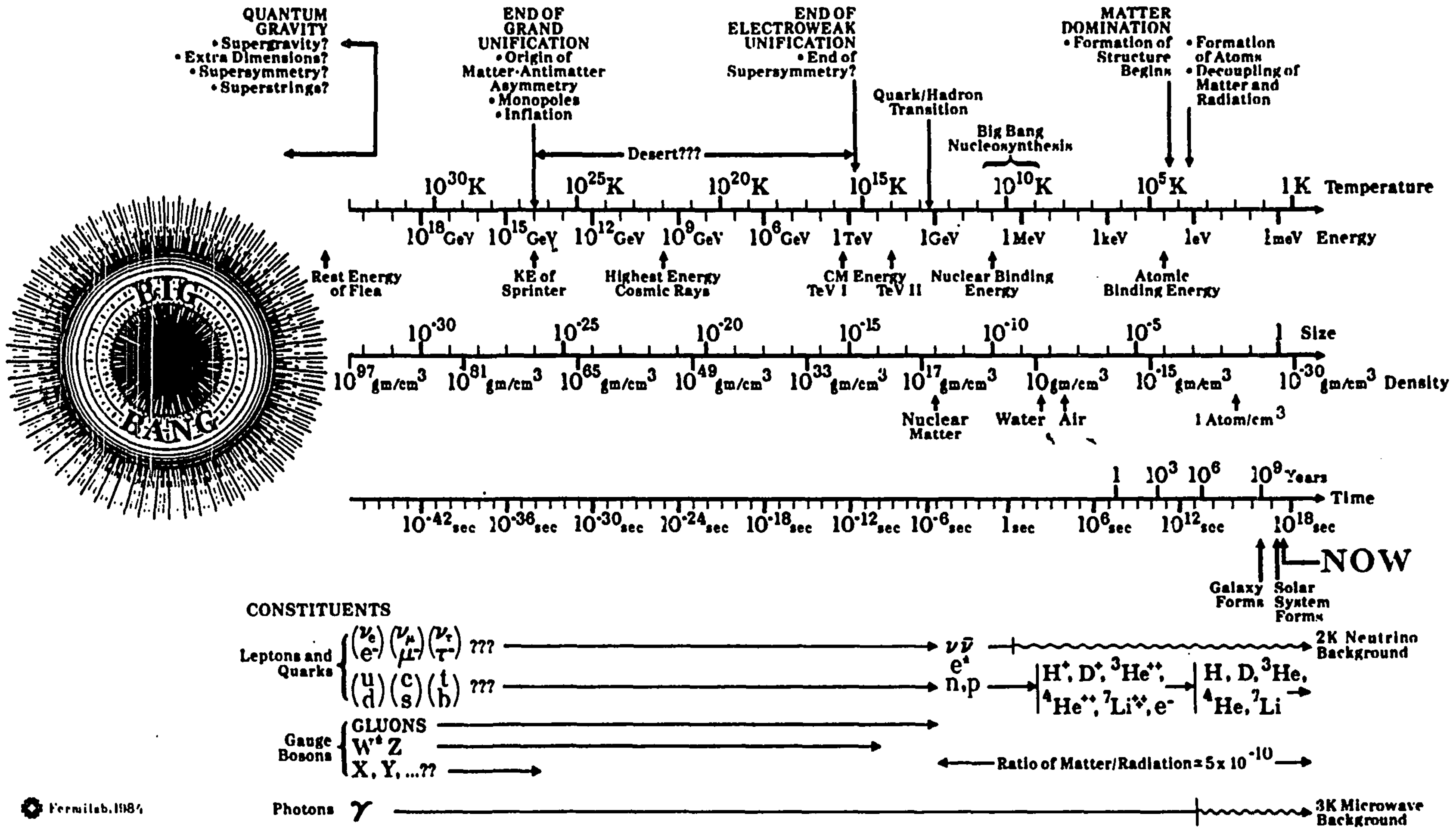
The issue whether or not the Universe is open or closed, i.e. whether it will continue to expand forever or at some point begin to contract, is still very much in debate. Certainly summing up all the "visible" matter of the

Universe does not result in enough matter being accounted for to close the Universe. Yet, there are galactic dynamical measurements which show that indeed not all the mass necessary to account for the known motions has been found. This is the famous missing mass problem. Numerous attempts are currently being made to try and account for this, many of them coming from the direction of high energy physics in the form of various particles going under the collective term WIMPs for weakly interacting massive particles. One of the chief candidates are the various neutrino species which, if the inflationary Big Bang scenario is right, then the Universe is positively filled with them and they would not need a very large mass to close the Universe. So far, the electron neutrino, seems to have an upper bound on its mass that probably rules it out. However, if neutrinos mix, then over galactic distances (probably very safe to be larger than the mixing length), all species of neutrinos may appear in equal numbers. There is a theoretical prejudice[1] which implies that the neutrino masses would roughly scale with the masses of their associated leptons so that

$$m_{\nu_e} << m_{\nu_\mu} << m_{\nu_\tau} \tag{4.1}$$

A very nice summary combining particle physics and cosmology is given in the following figure put out by Fermi Lab.

[1] This was introduced and used in Chapter 3 above.



QUANTUM GRAVITY
• Supergravity?
• Extra Dimensions?
• Supersymmetry?
• Superstrings?
END OF GRAND UNIFICATION
• Origin of Matter-Antimatter Asymmetry
• Monopoles
• Inflation
Desert???
END OF ELECTROWEAK UNIFICATION
• End of Supersymmetry?
Quark/Hadron Transition
Big Bang Nucleosynthesis
MATTER DOMINATION
• Formation of Structure Begins
• Formation of Atoms
• Decoupling of Matter and Radiation
10^30 K 10^25 K 10^20 K 10^15 K 10^10 K 10^5 K 1 K Temperature
10^18 GeV 10^15 GeV 10^12 GeV 10^9 GeV 10^6 GeV 1 TeV 1 GeV 1 MeV 1 keV 1 eV 1 meV Energy
Rest Energy of Flea
KE of Sprinter
Highest Energy Cosmic Rays
CM Energy TeV I TeV II
Nuclear Binding Energy
Atomic Binding Energy
BIG BANG
10^-30 10^-25 10^-20 10^-15 10^-10 10^-5 1 Size
10^97 gm/cm^3 10^81 gm/cm^3 10^65 gm/cm^3 10^49 gm/cm^3 10^33 gm/cm^3 10^17 gm/cm^3 10 gm/cm^3 10^-15 gm/cm^3 10^-30 gm/cm^3 Density
Nuclear Matter
Water
Air
1 Atom/cm^3
1 10^3 10^6 10^9 Years
Time
10^-42 sec 10^-36 sec 10^-30 sec 10^-24 sec 10^-18 sec 10^-12 sec 10^-6 sec 1 sec 10^6 sec 10^12 sec 10^18 sec
NOW
Galaxy Forms
Solar System Forms
CONSTITUENTS
Leptons and Quarks
(νe e-) (νμ μ-) (ντ τ-) ???
(u d) (c s) (t b) ???
ν ν̄
e±
n,p
2K Neutrino Background
H+, D+, 3He++, 4He++, 7Li+++, e-
H, D, 3He, 4He, 7Li
Gauge Bosons
GLUONS
W± Z
X, Y, ...??
Ratio of Matter/Radiation ≈ 5x 10^-10
Photons γ
3K Microwave Background
Fermilab. 1984

Figure 4.1 The history of the Universe according to standard cosmology and the standard model for fundamental particle physics

4.2 Cosmic-rays

In terms of the early twentieth century definitions of the different forms of radiation, cosmic-rays differ from the other forms of radiation such as α, β, γ, or X radiation in that they are not made up of a single component. The primary cosmic-rays, namely the composition of the extraterrestrial radiation that impinges upon the atmosphere of the Earth, is mainly made up of ions such as e^-'s, p^+'s, and heavier nuclei along with some neutral components such as n^0's, γ's, and ν's (the latter of possibly different families).

4.2a History and Importance

Cosmic-rays kept showing up as an interference in early radiation experiments leaving unexpected ionization in early detectors such as cloud chambers. They were shown to exist even underground through the seemingly spontaneous discharge of electroscopes. A further piece of the puzzle was unraveled when in 1912 Viktor Hess made a series of balloon flights which showed that this background radiation increased significantly with altitude.

Because one has no control over their generation, cosmic-ray studies have traditionally had to ride at the back of the high energy bus, so to speak (for at least the last 30 years or so), when compared to accelerator studies where the source is well known and well calibrated. Still, cosmic-ray studies have played an important part in the development of the high energy physics picture. It is in cosmic-rays that the first detections of the positron (e^+), the muon (μ), the pion (π meson), and some of the K mesons and hyperons were made. Because cosmic-rays are the only source for studying high energies > 1800 GeV, there is every reason to believe that in the future, when the development of particle accelerators mandates machines that want to circle the circumference of the Earth, there is every expectation that they will become even more important. Cosmic-rays even play a very important role in biological evolution on this planet for they are a primary source of ionizing radiation (the other being radioactive decay of terrestrial minerals), which is continuously needed in very low level quantities to induce random mutations at a rate that does not destroy life yet allows some regular change to be introduced so that the process of selective evolution can filter out the useful changes. This process is very evident when one looks at the evolutionary rates of alpine plants compared to plants closer to sea level. Radial distribution of plants, isolating populations on mountain tops, and then bombarding them with doses of cosmic-radiation above the sea level norm often produces unique species to a given mountain region. This is even true here on O'ahu island in Hawai'i where closely related but distinctly different species of ferns have evolved separately in the Waianae and Ko'olau mountain ranges.

Ultimately, cosmic-rays will hopefully tell us much more than just something about the properties of the particles involved but also much about their astrophysical sources. Of course, this makes for a very difficult experiment since there is no a-priori clean separation of phenomena between source and target as one has in an accelerator experiment where there is detailed knowledge and control of the beam. Thus it will take patience to sort it all out but astronomers have been doing this for years and compared to accelerator studies, cosmic-ray studies are very cheap.

In fact, cosmic-rays as high in energy as 10^{12} GeV have been detected. This is equivalent to about 10^{22} ^{0}K (right in the middle of the suspected particle desert between the Electroweak and Grand Unifications) corresponding to a time about 10^{-24} seconds after the Big Bang.

4.2b Secondary Components

Most of what is directly observed in terms of cosmic-rays are the secondary components. The *nucleonic* components are mostly what are called the knock-on p^+'s and n^0's. These are due to direct collisions by primary cosmic-rays or secondary π's of energies > 100 MeV resulting in nucleons being knocked out of atmospheric nuclei. Because of their charge, the p^+'s rapidly disappear due to ionization losses dominating over any nuclear interactions. This results in an excess of n^0's which at about an altitude of 3500 m above sea level is around 4 to 1. Ultimately these n^0's thermalize to around 0.025 eV.

When the nucleonic components are generated by sources with energies > 300 MeV, larger nuclear fragments such as D^+, T^+, and α particles maybe radiate out of the nucleus with typical energies around 10 MeV. These events appear on photographic emulsions as *stars* and are so named. Stars make up about 20% of the cosmic-ray secondaries and nucleonic components make up about 7% (near sea level).

The π^0 meson has the short lifetime of around 8×10^{-17} s decaying into two γ rays whose total energy equals the rest mass $m_{\pi^0} = 140$ MeV plus the kinetic energy of the π^0 meson. In turn, each γ ray, when it passes close enough to a nucleus, undergoes pair creation resulting in an e^+, e^- pair. In turn, these particles undergo Bremsstrahlung radiating lower energy γ rays which in turn undergo pair creation, etc. in an electromagnetic cascade until finally the individual particle energies are below the 1 MeV threshold for pair creation. With such cascades typically spending themselves at higher altitudes, these secondary particles are known as the *soft* components. About 25% of the secondary cosmic-rays (near sea level) are made up of soft components.

The charged π mesons are a different story. With a much longer lifetime of around 2.5×10^{-8} s before they decay into μ's by way of

$$\pi^{+,-} \dashrightarrow \mu^{+,-} + \nu_{\mu} \qquad (4.2)$$

those with an energy > 10 GeV are traveling fast enough to experience significant relativistic time dilation. Thus, they stay around long enough to interact with atmospheric nuclei much in the same way that primary cosmic-ray nucleons do with their cross section of interaction being essentially the geometric cross section of the nuclei involved. When there is decay and the μ's are produced, the μ's do not interact with the nuclei. Additionally, because they are so massive ($m_{\mu} = 207\ m_e$), they fail to produce Bremsstrahlung so their only energy loss is through the relatively feeble process of ionization. As such, μ's can be quite penetrating going to great depths in rock or the ocean and are known as the *hard* component of cosmic-rays making up about 48% of the secondary cosmic-ray spectrum (at sea level).

4.3 Cherenkov Radiation

In 1934 the Russian Physicist Pavel A. Cherenkov[2] observed a form of electromagnetic radiation that is today named after him. It occurs when a charged ultrarelativistic particle passes through a material medium. It is a necessary condition that[3]

$$v > \frac{c}{\sqrt{\varepsilon(\omega)}} \tag{4.3}$$

where v is the speed of the particle, c is the absolute speed of light (in a vacuum), and $\varepsilon(\omega)$ is the dielectric constant of the material medium. In other words, the speed of the particle is greater than the local speed of light (phase velocity of the electromagnetic fields at frequency ω). When such a condition is met, radiation is emitted with a characteristic angle of emission θ_c to the direction of travel, forming a light cone, given by

$$\cos\theta_c = \frac{1}{\frac{v}{c}\sqrt{\varepsilon(\omega)}} \ . \tag{4.4}$$

Following the experimental observation of the (electron) neutrino, Fred Reines proposed the use of the Cherenkov effect in the deep ocean as a means of observing Cosmic Ray neutrinos. The effect of the ocean would be two-fold. First, by going deep and aiming the Cherenkov light detector downwards so that the earth worked as a shield against other high energy Cosmic Rays, an unambiguous neutrino detector could be built. Second, since neutrinos are notoriously hard to detect because of their small interaction cross-section, such an ocean based detector is essentially infinitely expandable and so could be built up to sufficiently large detector area as to be able to detect a signal from astrophysical sources. This idea

---

[2] receiving the 1956 Nobel prize in Physics for his observation

[3] see Jackson, p. 638-9

goes by the acronym of **DUMAND** for Deep Underwater Neutrino and Muon Detection. Neutrinos from such sources would come from very high energy astrophysical processes indeed and as such would potentially reveal much information about the dynamics of such processes. The only thing that has kept 20th century science from pursuing this idea (from the 1950's) has been that it was not until very recently that the technology has existed in order to be able to do this.

4.4 DUMAND and the SPS

The technologies needed to do this, besides the associated marine engineering (which is pretty involved in its own right), are those of large photomultiplier tubes of 5 ns resolution, fast digitizing electronics of 5 ns resolution, fiber optics communications, microprocessor controllers, and high pressure (> 500 atmospheres) housings. These all had to be integrated into a working instrument. The project which did this, during a cruise off the Kona coast of the island of Hawai'i in the Fall of 1987 is known as the **SPS** for Short Prototype String.

Chapter 5 is a review of the classic experiments that have taken place in the intervening decades leading up to the technological feasibility of such an endeavor and as such many associated details are left for there. Here, a over view of the SPS will be given. Chapters 6 through 10 deal with various aspects of the instrument design that I was involved in, my particular emphasis being the microprocessor controller technology, both hardware and software. Finally, chapter 11 is an analysis of some aspects of the data from the SPS project concluding that a larger DUMAND array is technically possible.

The SPS in part deserves its name because geometrically it is a string of photomultiplier tube detector based modules that are housed in transparent (machined pyrex glass) pressure housings capable of handling the deep ocean pressures involved. A cross sectional diagram of such an optical module is shown below in Figure 4.2:

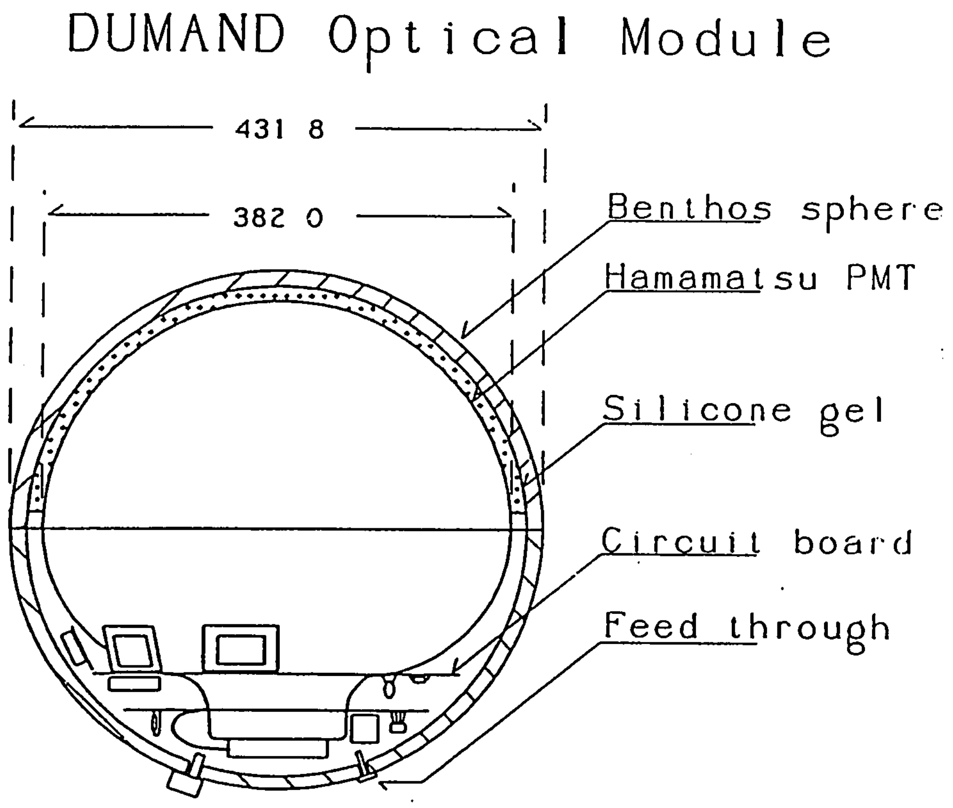


Figure 4.2 Cross-sectional diagram of an optical module found in the SPS[4].

[4] see Matsuno, et. al. 1988

Note that the photomultiplier is big in the sense that it is 16 in in diameter and that the pressure housing is 17 in in diameter. The photomultiplier has a resolution of 5 ns and a gain of 10^7. The electronics to control, power, and process the signal from the photomultiplier are all limited to the small space found around the base of the tube. For the purposes of the SPS and its goal of demonstrating technical integration feasibility, in contrast to any permanent DUMAND type facility where the principle goal would be to detect muons induced by Cosmic Ray neutrinos, the photomultiplier tubes were oriented upwards rather than downwards so that they would be most sensitive to the more abundant atmospheric muons thus greatly enhancing the rate at which the SPS telescope would have to reliably detect a muon track. In total, seven such optical modules were strung together along with two calibration modules and additional instruments to monitor the deep ocean environment in a detector string shown in Figure 4.3 below[5]:

[5] see Figure 4.1 Bosetti, et. al. 1988 (DUMAND II Proposal)

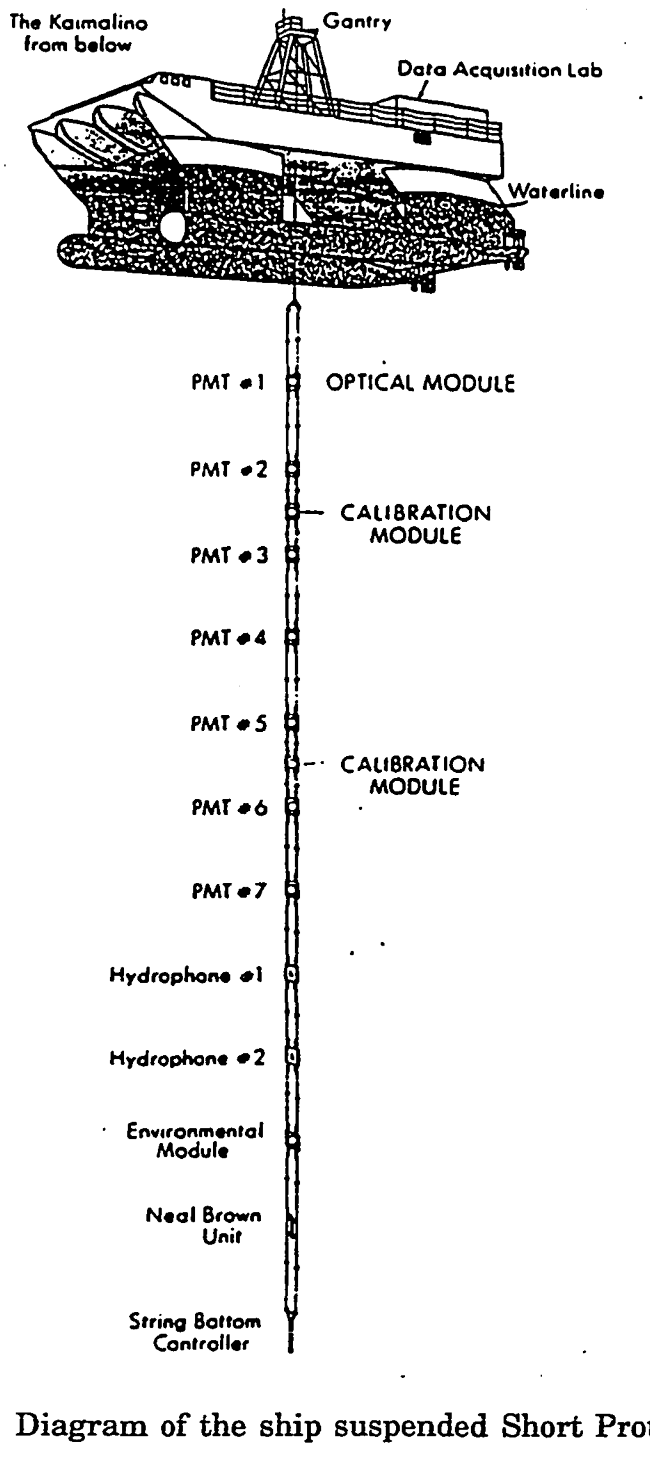


Figure 4.3 Diagram of the ship suspended Short Prototype String experiment.

At the speed of light (equivalent to 0.333 ns m^{-1} at *c*), a 5 ns resolution yields a maximum travel distance of 1.5 m (actually a bit less for light slowed in water). Thus, to be on the safe side, the optical modules had to be spaced at a distance several times greater than this. Dimensionally, the instrument string was about 50 m vertical with optical modules spaced at 5.1 m intervals (equivalent to about three to four times the minimum allowable distance). With 5 ns resolution, the intent was to be able to detect a signal through time of flight separation of the individual photomultiplier detectors. As such, the path of the relativistic muon (neutrino induced or otherwise) should be reproducible to a cone about the axis of the string and its direction unambiguously determined. As you will see in the review of experiments in the next chapter, this is a considerable improvement over previous experimental arrangements after several decades of trying on the part of the world physics community.

As Figure 4.3 shows, the string consisted of a specially designed 7.9 mm diameter electro-optical cable which was suspended from the SSP Kaimalino[6] whose particular characteristics plus the cable rigging on board tended in combination to minimize any accelerations along the cable cutting back on mechanically stimulated bioluminescence (the other major background source being K^{40} decay which is beat down through track fitting). This cable had three functions to perform. One, it was the mechanical link between the instrument string and the ship. Two, it was the optical link for the high speed data being sent to the ship board laboratory. And, three, it was the electrical conductor for both instrument power as well as slow speed command and control communications.

As Figure 4.3 shows, the calibration modules were placed half way between the 2nd and 3rd and the 5th and 6th optical modules. These consisted of a UV laser source of known and programmable intensity, duration, and repetition whose light activated a scintillator at the end of an extended rod (the "dork") emitting a blue light pulse close to the maximally transparent

[6] Hawaiian for "calm seas"

region of water. With such pulses programmed into the data stream, absolute calibrations were possible. Finally, in anticipation of a permanent ocean anchored array of DUMAND vertical strings, an environmental module along with two associated hydrophones (in the hopes that one may see one's way clear to doing an "acoustic" version of DUMAND) and a "Neal Brown" deep ocean sensor unit for monitoring the deep ocean environment were deployed. The outputs of all these devices were in turn fed to the String Bottom Controller (SBC) whose principle task was to convert parallel fiber optic analog inputs into a single serial fiber optic digital output which constituted the high speed data stream sent back up the fiber optic cable to the ship board laboratory and its data harvesting and system command and control computer.

CHAPTER 5 – The State of the Art – A Review of Previous Experimental Investigations

With the discovery in 1962 of a neutrino species associated with the muon, the possibility of detecting high-energy neutrinos from primary cosmic rays and ultimately interpreting the information that they may convey about their source became real. The consequence of this was that a series of cosmic ray detection experiments were proposed, built, and run in the late 1960's through the middle of the 1980's. In this section, a comparative description of a number of these experiments will be made as motivation leading up to the development of the DUMAND Short Prototype String (SPS).

Atmospheric muons are very penetrating and have a rate something on the order of 10^{10} times greater than the expected rate of muons from cosmic ray neutrinos at a depth even as great as several kilometers of water. To detect such a weak signal against such a strong background requires an experimental procedure that greatly cuts out the atmospheric muons. Two approaches are possible, (1) seek a great depth and use a relatively simple detector to reduce the atmospheric muons or (2) seek a moderate depth and use a more involved detection scheme whereby only upward traveling muons, i.e. muons that are unambiguously the product of neutrinos that traveled through most of the Earth until they interacted with the material medium near the detector, are accepted. The former requires depths that are impractical (greater than three kilometers in mines and ten kilometers in water) so that the latter is the only practical approach.

These experiments are divided into two basic detector categories. The earlier experiments, namely the Kolar Gold Field, Case-Witwatersrand-Irvine, LSD, Artymovsk, and Baksan experiments were all scintillation counter experiments. The later experiments, namely the Irvine-Michigan-Brookhaven and Kamiokande experiments are water

Cherenkov experiments. These and other recent experiments (Soudan, NUSEX, MACRO, LVD) were built primarily for other purposes, such as the search for nucleon decay or magnetic monopoles, but of necessity are excellent neutrino detectors as well. A number of experiments based upon both methods of detection will be reviewed here. However, it is important to note that we are only interested in the higher energy neutrinos and as such low (MeV to GeV) energy neutrino interaction detectors which look for contained events such as Artymovsk, LSD, LVD, the work of Davis, Lande, etc. will not be reviewed but rather only experiments that deal with through going muons will be considered.

5.1 Description of the Underground Scintillator Experiments

When an ionizing particle passes through a material medium, some of the atoms in the medium absorb energy in such a way that outer shell electrons may be pumped into a higher energy orbit. Some time later, these electrons return to their normal ground state sometimes radiating photons which are detectable by very sensitive light detectors such as photomultipliers. This is the phenomenon of scintillation. Typical inorganic phosphors (scintillators) such as ZnS used in early nuclear work by Rutherford have response times (the time between stimulation and return to ground state) of around 100 ns or longer. For detecting cosmic rays moving roughly at the speed of light (0.3 m ns^{-1}), in detectors with dimensions of a few meters, this does not give sufficient resolution to discern upward from downward motion. The development of organic liquid scintillators with a response time around 1 ns partially solved that problem and permitted the development of improved detectors.

However, in general, these detectors could not tell the difference between left and right nor up and down and at best, as in the case of CWI and KGF (described here below), where direction determination is possible, one is left with a two fold directional ambiguity unless there is an obvious intruder revealing the direction sense. Generally, liquid scintillators permitted the development of detectors with a larger detection area for a smaller cost. Only the large water Cherenkov type detectors such as Kamiokande and IMB are to date able to distinguish the direction by timing as well as by topology. Let's look at some of the detectors and their evolutionary history.

5.1a Kolar-Gold Field

The Kolar-Gold Field experiment was a collaboration of the Tata Institute of Fundamental Research (India), Osaka City University (Japan), and the University of Durham (United Kingdom). The basic philosophy of the experiment was to use a simple detector at great depth. It was located at a depth of 7600 feet in the Kolar Gold Mines of South India. The experiment was carried out from early 1965 to June 1969 in basically three stages. The first stage consisted of muon telescopes as shown in Figure 5.1.

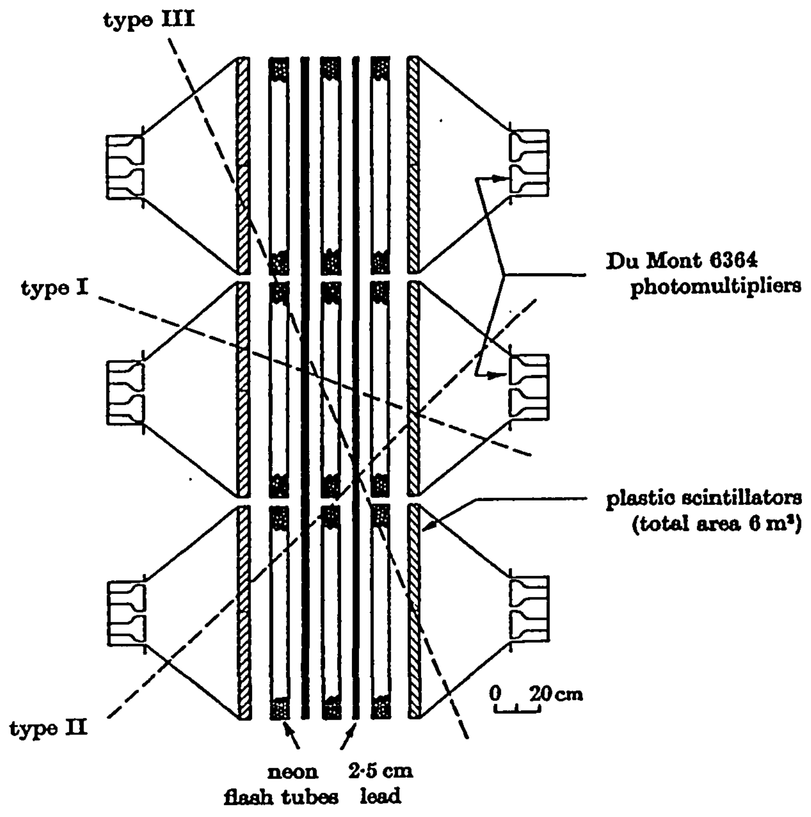


Figure 5.1 Geometrical arrangement of the first stage design (Kolar-Gold Field telescopes 1 and 2)[1].

[1] Menon, M. G. K., et al (1967)

The system was made up of two such identical telescopes each consisting of two vertical walls of plastic scintillators separated by 80 cm each 2 m long and 3 m high. The scintillator walls were equally divided up into three sections each with an effective area of 1 m^2. Each scintillator section was viewed by adjacent 5 in. diameter photomultipliers. Four fold coincidences were recorded between a pair of photomultipliers on one wall and any pair on the opposite wall. The space between the scintillator walls was occupied in part by three separate arrays of (horizontal) neon flash tubes each containing four columns of tubes. Separating the flash tube arrays were two walls of lead absorber each 2.5 cm thick.

Upon a four fold coincidence, the photomultiplier pulses were recorded on oscilloscopes and after about a 30 microsecond delay, a high voltage pulse was applied to the electrodes of the neon flash tube arrays. The flashes were then recorded photographically by two cameras in direct view of the flash tubes.

This arrangement verified the existence of nonelastic cosmic-ray neutrino interactions but not much else. The neon flash tubes were 200 cm in length and 1.8 cm in diameter. Intrinsic to this arrangement was a resolution of around 1 degree for zenith angle but very poor resolution for the azimuthal angle since it could only be approximated by noting which scintillators were involved in an event.

The second stage was developed to improve on the azimuthal uncertainty and three identical muon telescopes of a newer design were added as shown in Figure 5.2.

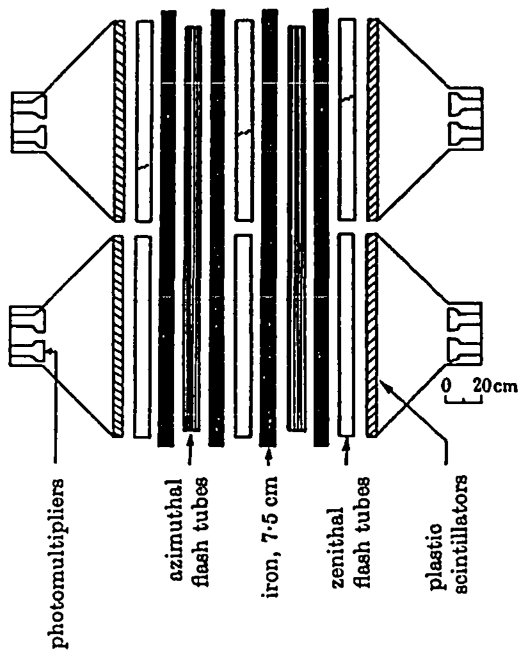


Figure 5.2 Geometrical arrangement of the first stage design (Kolar-Gold Field telescopes 3, 4, and 5)[2].

Each telescope consisted of two vertical walls of plastic scintillators separated by 130 cm each 4 m squared in area. The scintillator walls were equally divided up into two sections. Each scintillator section was viewed by adjacent pairs of 5 in diameter photomultipliers. The absorber was made up of four 7.5 cm thick iron walls placed between the flash tube arrays. Both zenithal (horizontal) and azimuthal (vertical) flash tubes were used with four layers of tubes in each array. Using mirrors, a single camera photographs simultaneously all five flash tube arrays. With the addition of azimuthal flash tubes, the azimuthal uncertainty was reduced to about one degree.

[2] Menon, M. G. K., et al (1967)

Initially, triggering was done the same as in telescopes 1 and 2 on a four fold coincidence of any two photomultiplier tubes on each side of the telescope. These telescopes were run this way from March 1966 when they were commissioned until early 1968. At this time the triggering was changed to being any pair of photomultiplier tubes on one (either) side of the telescope (o.s.t. or one sided triggering). This results in many more false triggers but its key advantage was that it permitted the detection of low energy events in which the particle did not penetrate all the way through the instrument thus permitting the neutrino interactions within the telescope to be detected as well as opening up the aperture of the telescope.

The third and final stage saw the addition of two identical spectrographs to help improve the accuracy of the azimuthal angle as well as to enhance the chances of distinguishing between muons and pions. Each spectrograph consisted of two scintillator walls 2 m high and 4 m long separated by 100 cm and containing as in the original two telescopes, horizontal flash tubes. A 40 cm thick iron block in the form of a central magnet was used as a absorber in each of the two spectrographs as shown in Figure 5.3.

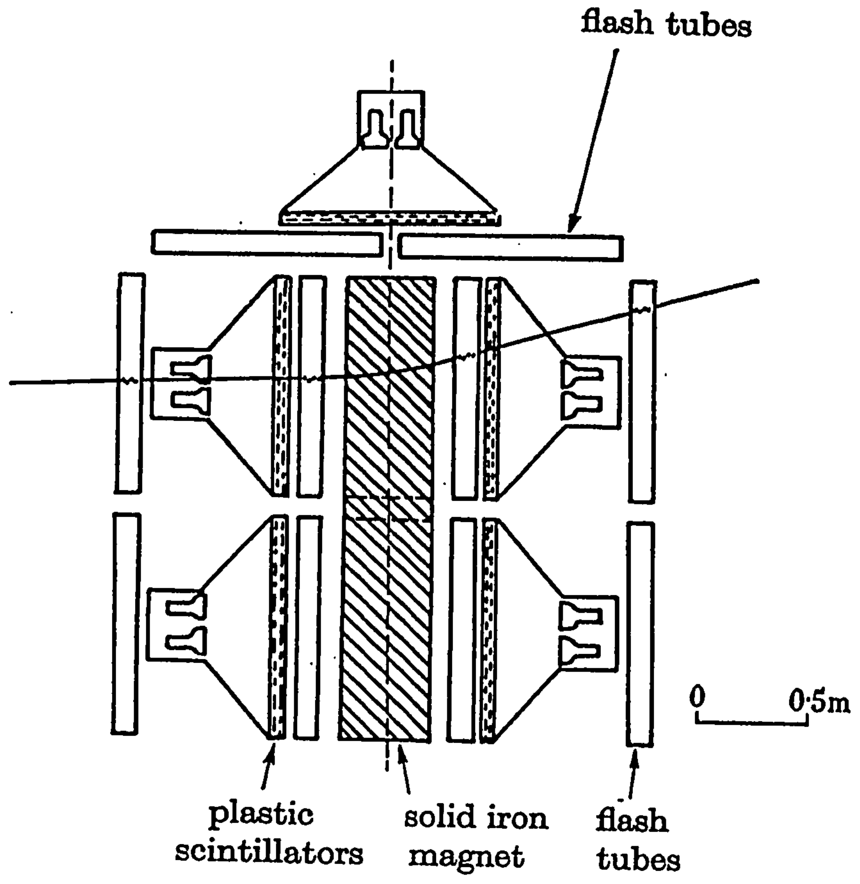


Figure 5.3 Geometrical arrangement of the third stage design (Kolar-Gold Field spectrographs 1 and 2)[3].

The absorber layers provided a means by which electrons could be distinguished from heavier particles such as muons and pions. Also, studying the production of secondaries from interactions within the absorber allowed the determination of the sense of direction of the incident particle.

[3] Menon, M. G. K., et al (1967)

Figure 5.4 shows the final arrangement of telescopes and spectrographs lined up along an East-West axis horizontally in a tunnel at the 80th level of the Heathcote Shaft in the Champion Reef Mines of the Kolar Gold Mining Undertaking.

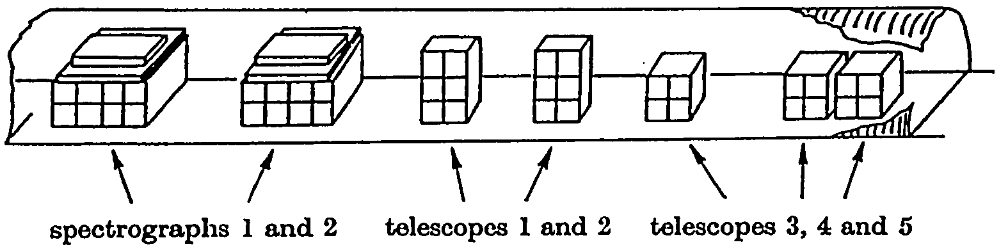


Figure 5.4 Final arrangement of the instruments in the Neutrino Experiment at the Kolar-Gold Field[4].

Neutrino events were distinguished from atmospheric muon events by making the crude cut of a zenith angle of 50 degrees. Anything greater than this was assumed to be an atmospheric muon and anything below this was taken to have passed through so much overburden of rock that it had to be that much of the path was traversed as a neutrino before interaction gave rise to detectable muons. Table 5.1 summarizes the experiment's published events:

4 Krishnaswamy et. al 1971a

Table 5.1

Summary of events in the Kolar-Gold Field experiment[5].

TABLE 1. DIVISION OF EVENTS AND EXPOSURE TIMES

	tels. 1, 2	tels. 3, 4, 5	o.s.t.	specs. 1, 2
exposure time/h	49047	17560	26196	24813
number of atmospheric muons	42	2	76†	82
number of neutrino-induced events ($\phi \geqslant 50°$)	7	2	5	2
aperture × time appropriate to neutrino-induced events/m^2 s sr ($\phi \geqslant 50°$)	2.10×10^9	0.37×10^9	0.85×10^9	1.39×10^9

† Includes particles passing through only one tray of flash-tubes.

[5] Krishnaswamy et. al. 1971a

Finally, the first ever attempt to possibly associate neutrinos with extraterrestial sources was made by projecting the directions of the neutrino events back onto the Celestial Sphere.

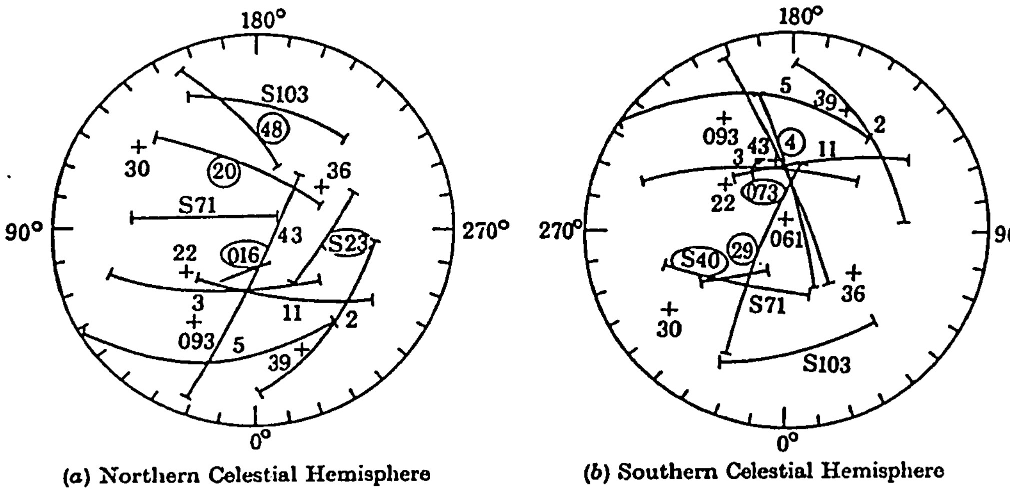


Figure 5.5 Projection of the detected neutrino events in the Kolar-Gold Field experiment back onto the Celestial Sphere[6].

[6] Krishnaswamy et al 1971a

5.1b Case-Witwatersrand-Irvine

The Case-Witwatersrand-Irvine (CWI) experiment[7] was a collaboration of Case Western Reserve (USA), the University of the Witwatersrand (South Africa), and the University of California at Irvine (USA). Like the KGF experiment, the basic philosophy was one of building a simple detector and placing it at great depth. It was located at a depth of around 2 miles (10,778 feet in the second stage) where the virgin rock temperature is reported to be 123 degrees Fahrenheit. The experiment was carried out from mid 1964 to October 1971 in two distinct stages.

The first stage consisted of an array of large ultraviolet transmitting lucite tanks filled with a mineral oil based liquid scintillator. The tanks were grouped in "bays" of six each making up two parallel walls of three tanks. The three tanks in each wall were stacked one on top of the other with the one on the bottom designated L for lower, the one in the middle M for middle, and the one on the top U for upper as shown in Figure 5.6.

[7] Reines, F. (1967)

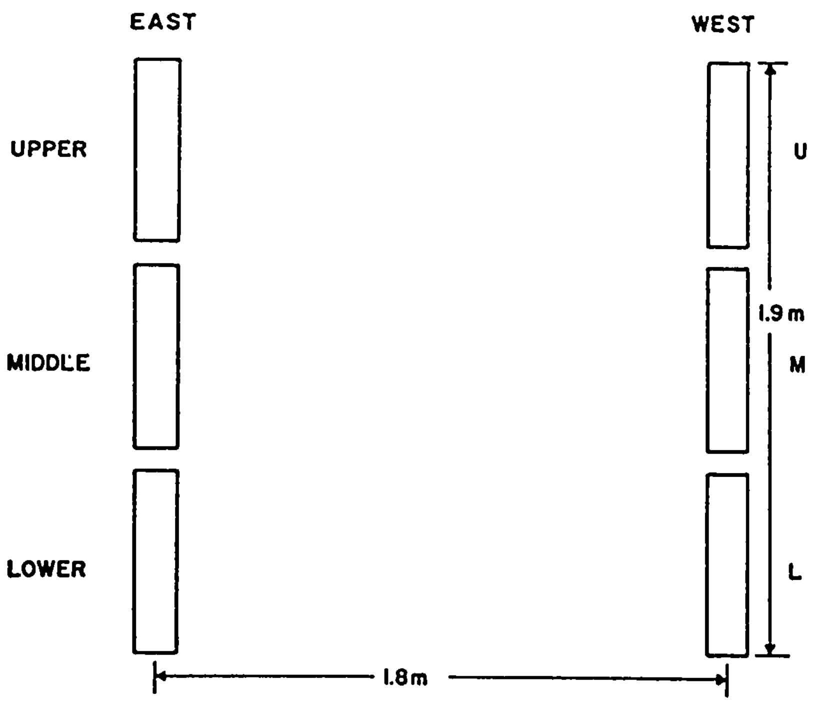


Figure 5.6 Cross section of CWI stage I array[8].

The entire array consisted of nine such bays mounted on two parallel and discontinuous rails along a mine tunnel 3 meters by 3 meters in cross section and 150 meters long. The basic orientation of the tunnel was along a North-South axis so that the detector was most sensitive in the East-West direction. This resulted in a detector array made up of a total of 54 scintillation detector elements covering a total effective area of 160 meters squared as shown in Figure 5.7.

[8] Reines et. al, 1967

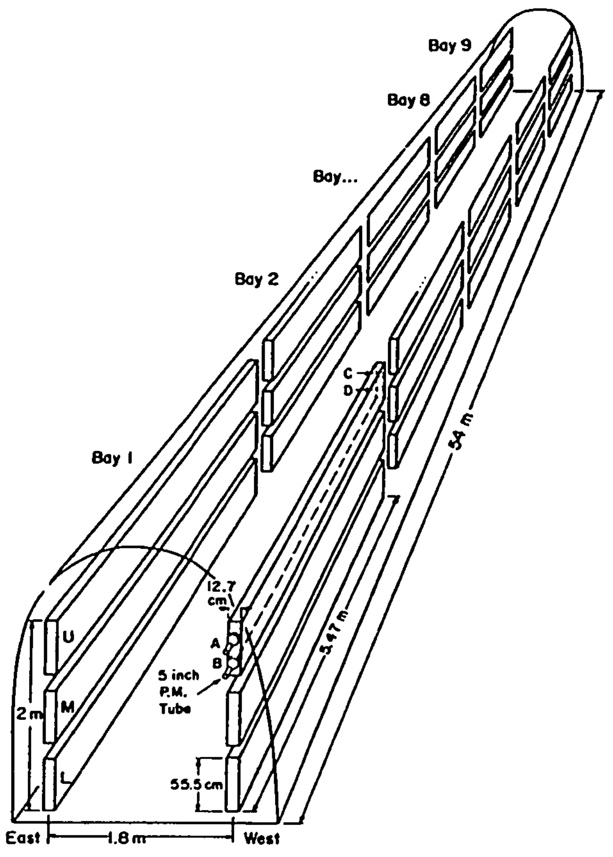


Figure 5.7 Perspective sketch of the CWI stage I array[9].

The individual detector elements, one of which is shown in detail above in Figure 5.7 were 5.47 m long, 12.7 cm wide, and 55.5 cm high. Each end of the detector elements had two 5 in photomultiplier tubes facing into the liquid scintillator (the tube pairs being labeled A and B on one side and C and D on the other side). This arrangement was chosen to satisfy the following criteria (Reines 1971):

[9] Reines, et. al. 1971

(1) a large and relatively inexpensive surface area viewed by a small number of photomultiplier tubes,

(2) a thickness sufficient to ensure energy deposition by a penetrating charged particle well in excess of that due to natural radioactivity,

(3) a height consistent with the tunnel dimensions and the desired hodoscope angular resolution,

(4) a response function such that pulse height variations over the length of the element were not excessive.

Careful study with identical scintillator elements on the surface of the Earth, selecting atmospheric muons with known paths by means of two small guide detectors, produced the response function (the relative signal amplitude seen by a photomultiplier as a function of the event location). This permitted identification of the center position of the scintillation along the active length of the detector element to within an uncertainty of $\pm$ 0.15 meters. Thus, with events in which multiple detector elements were involved, approximate event tracks could be reconstructed. Figure 5.8 shows this response function.

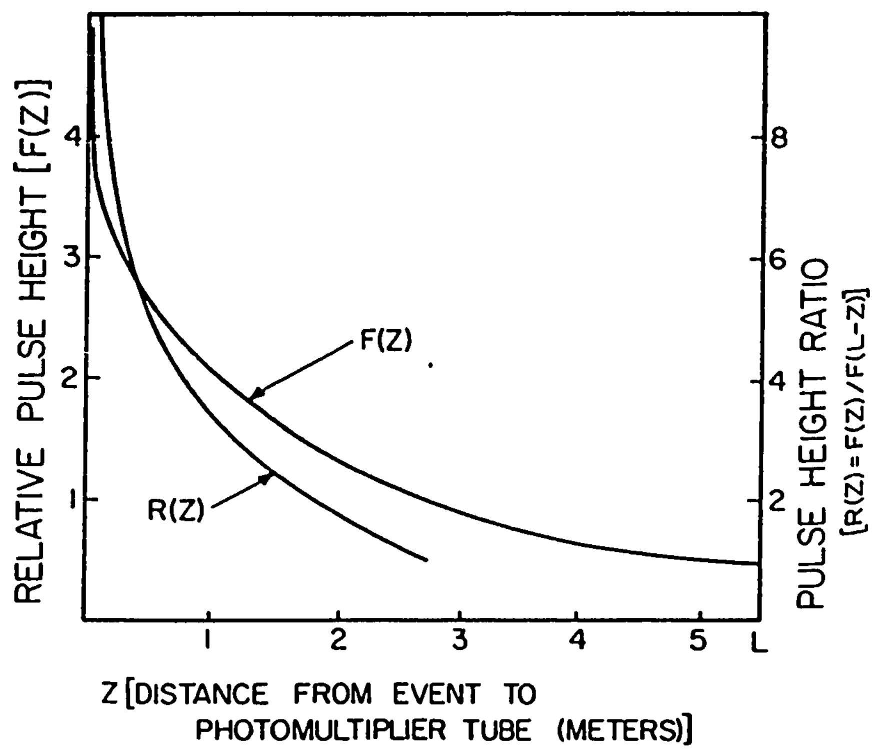


Figure 5.8 The response function of a detector element in the stage I array of the CWI experiment[10].

Triggering was accomplished upon a four fold photomultiplier coincidence and the individual charges were stored on capacitors in an arrangement called the "chronotron" because it preserved the time sequence of events for later readout. With this geometry, three angular ranges on each side of the horizontal, 0 - 20 degrees, 20 - 40 degrees, and 40 - 50 degrees could be established. This experiment verified the presence of an isotopic muon flux due to muons (produced by neutrinos interacting with the surrounding rock which in turn were produced in the atmosphere) as well as a sharply peaked flux of atmospheric muons that could penetrate to depth around a small zenith angle. The data was found to be consistent with several maximum likelihood studies yielding a total

[10] Reines 1971

rate for neutrino induced muons of $(6.5 \pm 1.1) \times 10^{-7}\ s^{-1}$ for an isotropic neutrino flux of $(3.7 \pm 0.6) \times 10^{-13}\ cm^{-2}\ s^{-1}\ sr^{-1}$.

Somewhat akin to what was done with the later stages of the Kolar-Gold Field experiment, the second stage of this experiment included crossed flash tubes to try and pin down the angular distribution better.

The second stage of the CWI experiment was set up in another mine tunnel about 50 meters deeper at a depth of 10,788 feet or under $8.89 \times 10^{+5}$ g cm^{-2} of standard rock. This stage of the experiment, called the ERPM (East Rand Proprietary Mine), operated from Dec. 13, 1967 until Oct. 28, 1971.

Like the first stage, the second stage consisted of an array of large ultraviolet transmitting lucite tanks filled with a mineral oil based liquid scintillator. The tanks were grouped in "bays" of three each making up a wall three tanks high. At one end, eight bays arranged to form two parallel walls as in the first stage were set up to permit direct comparison to the first stage experiment. Each bay was surrounded by a criss-crossed pattern of flash tubes totaling 48,384 tubes in all. The total effective area for this array was 174 m^2. This is all shown in Figure 5.9 below.

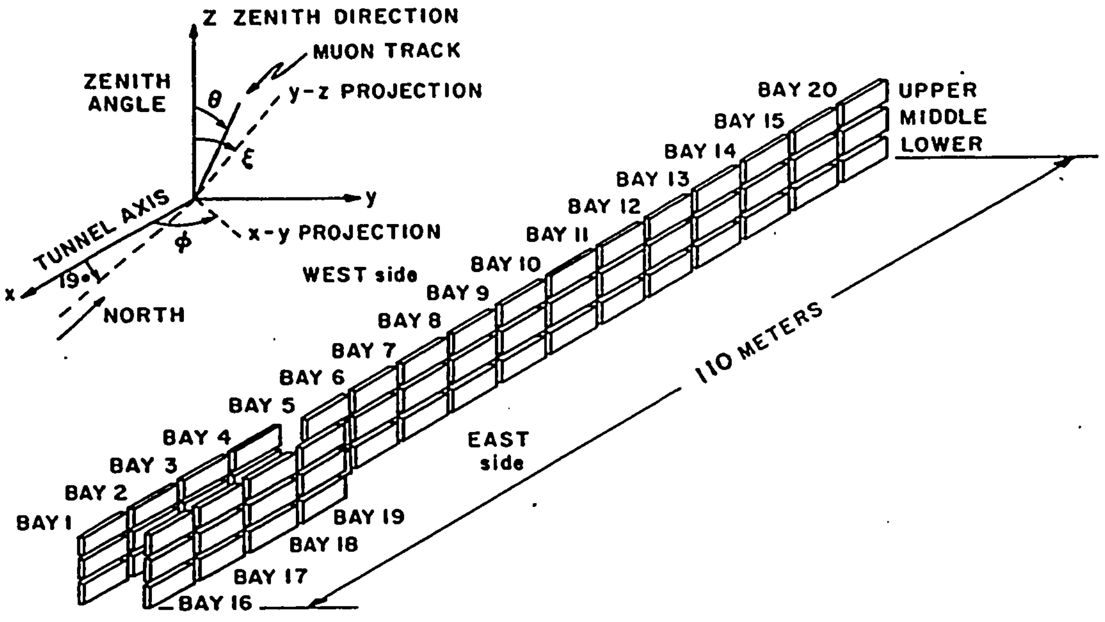


Figure 5.9 The geometrical layout of the CWI stage II array[11].

The flash tube array was constructed such that 56 tubes in two layers comprised an "element" 0.5 m x 2.0 m in area. 18 such elements made up a "module" of six horizontal and six vertical layers extending over a region the size of 1/3 of a bay. Finally, three such modules covered a bay in a criss-cross pattern of flash tubes yielding an effective zenith angle distribution of a few degrees markedly improving the resolution.

[11] Crouch, 1978

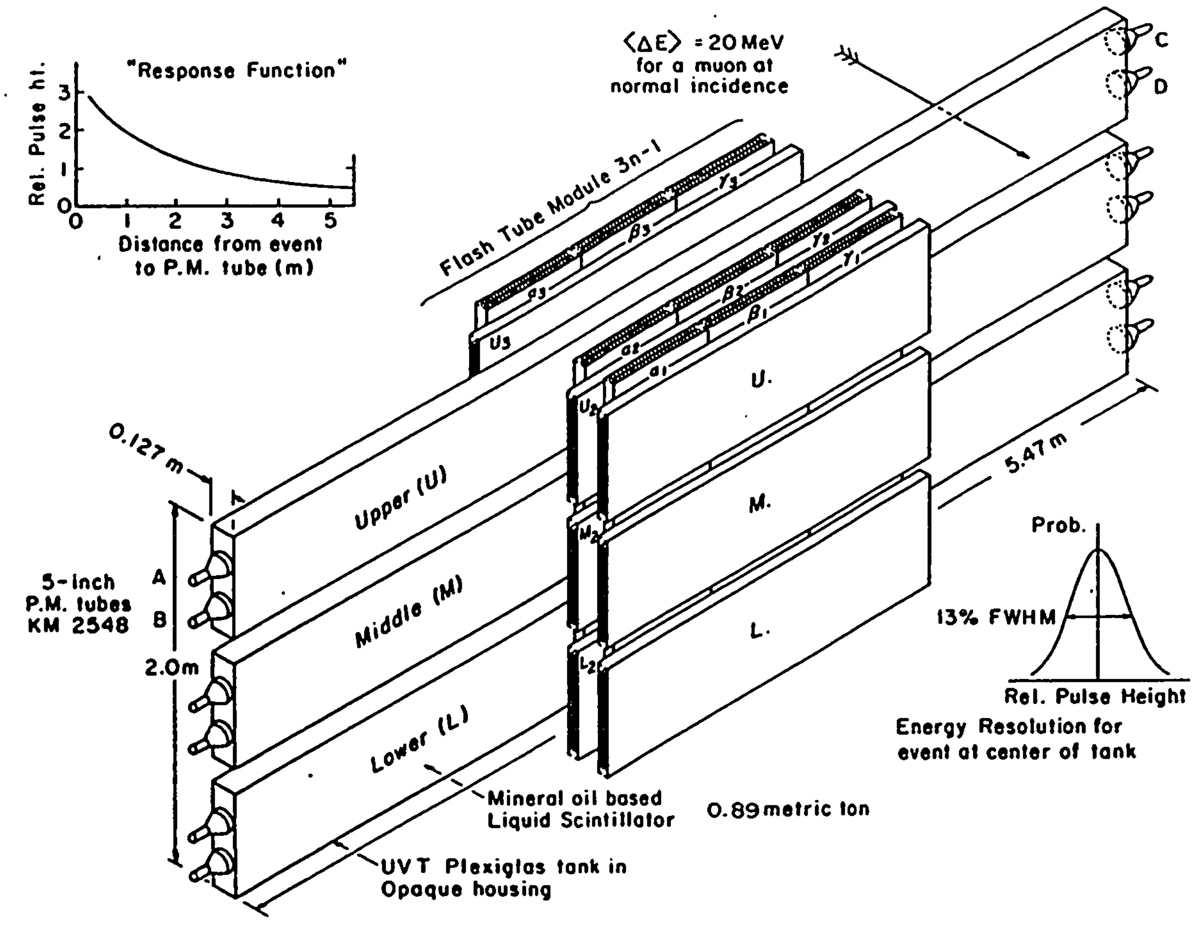


Figure 5.10 Details of the nth scintillator bay and the (3n -1)st flash-tube module in the stage II array[12].

[12] Crouch 1971

Triggering was again based upon four fold coincidences but with the crossed flash tube arrangement there was no real need for concern about such things as "response functions". However, this experiment, like the Kolar-Gold Field suffered from the inability to distinguish unambiguously the direction of events along a given track.

Analysis of the data showed indeed an isotropic distribution of muons attributed to atmospheric neutrinos interacting within the surrounding rock. Unambiguous neutrino events were taken as large zenith angle events (> 45 degrees). This flux approximated the predicted value so that the muon flux due to extraterrestrial neutrinos, taken as no more than 20 % of the observed horizontal flux or $< 10^{-13}$ cm^{-2} s^{-1} sr^{-1} should the neutrino spectrum have the same slope as the atmospheric spectrum. The isotropic term gave an atmospheric neutrino flux of $(2.23 \pm 0.20) \times 10^{-13}$ cm^{-2} s^{-1} sr^{-1} , an experimental value not improved upon for nearly 20 years.

5.1c Baksan

Yet another attempt to explore the realm of muons and neutrinos deep underground through the use of liquid scintillators is the Baksan Underground Scintillation Telescope located at the Baksan Neutrino Observatory in the North Caucasus in the Baksan Valley at an altitude of 1700 m. This experiment is the product of a single group at the Institute for Nuclear Research of the USSR Academy of Sciences in Moscow. Unlike the previous two experiments, it is not located in a deep mine as much as it is buried inside of a mountain thus having the relatively shorter rock overburden of 850 hg cm^{-2} and so requiring a more complex detector to unambiguously separate out the upcoming neutrinos (see A in Figure 5.11 below). The experiment, begun in the late 1970's is still operational.

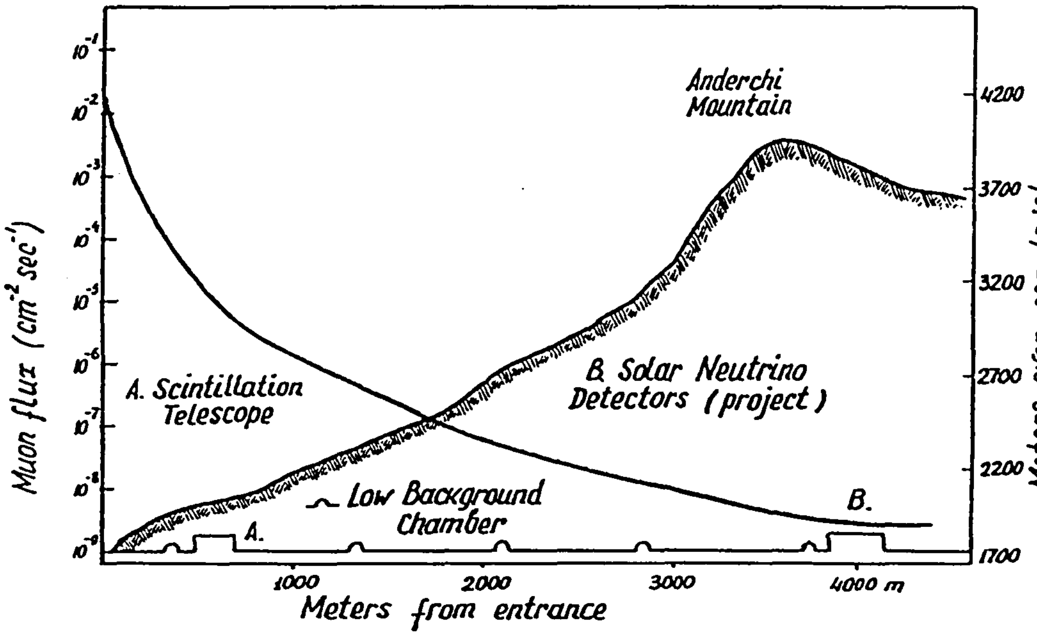


Figure 5.11 Diagram showing approximate location of the scintillation telescope in its relationship to the mountain above it and to a nearby solar neutrino detector[13].

[13] Alekseyev, et. al., 1979

The basic geometry, consists of eight working surfaces populated with 3132 (or more recently reported 3156) "standard" (liquid scintillation) detectors. Four of these surfaces are horizontal (two interior and two exterior) and four are vertical (all exterior) arranged in a sort of semi-cube as shown in Figure 5.12. The overall dimensions of the semi-cube are 16 m x 16 m x 11 m high with a total target mass of 330 tons.

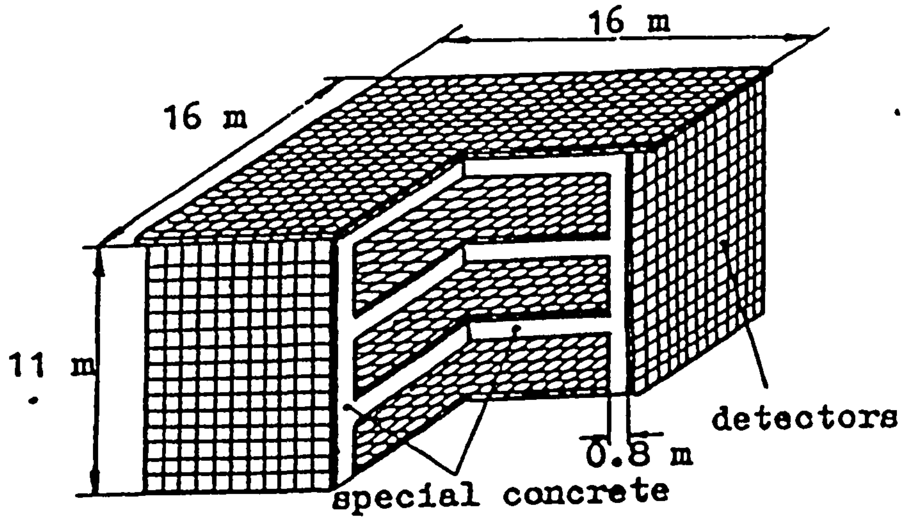


Figure 5.12 Cross-sectional drawing of the Baksan telescope[14].

The individual standard detectors consist of an aluminum tank, 70 cm x 70 cm x 30 cm, filled with oil based liquid scintillator and viewed by a single 30 cm photomultiplier tube.

[14] Alexseyev, et. al. 1987

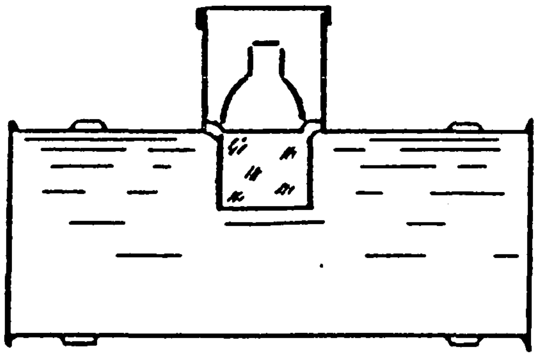


Figure 5.13 The "standard" scintillation detector of the Baksan telescope[15]

The stated goals of the triggering system for recording upward particles are three in number:

(1) cover most of the solid angle region below the horizon,

(2) respond fast enough so that the pulse shape can be recorded on a multitrace oscilloscope, and

(3) cut off the downward flux as much as possible.

A combination of three triggering systems is used to accomplish this covering a zenith angle range from 80 to 180 degrees with time of flight being used to distinguish real events. Thus, this telescope differs from the KGF and CWI arrangements in that it is not limited as much to a region around the horizon as they are for exploring unambiguous neutrino signals. Azimuthal direction should also be determined. The reported effective area of the instrument is 100 m^2. The papers reviewed here do not yet report out an atmospheric neutrino flux in a referenced journal.

[15] Alexeyev 1979

5.2 Description of the Underground Cherenkov Experiments

When a charged particle moves through a material medium faster than the local speed of light in that medium, a shock wave is set up analogous to the sonic boom heard in air when an object moves through the air at a speed greater than the speed of sound. This is the phenomenon known as Cherenkov radiation. The cosine of the angle of the resultant (shock) light cone is the inverse of the index of refraction in the transparent medium. For pure water, with an index of refraction of $n = 1.33$, this corresponds to a Cherenkov angle of 41.4 degrees. Like scintillation, Cherenkov light is detectable with very sensitive light detectors such as photomultplier tubes but such arrangements usually have less available light to detect on the order of a factor of ten or more. The trade off with the loss of light is that the Cherenkov technique is potentially highly directional.

One prediction of supersymmetric theories was that the proton decays (diamonds are not forever) with a half-life on the order of 10^{29} to 10^{33} years assuming a simple SU(5) symmetry. To check this prediction, one could build a detector that sat and looked at one proton for 10^{33} years (which is many orders of magnitude beyond the age of the Universe $\sim 10^{10}$ years) or one could build a detector that looks at 10^{33} protons over the course of a year or so. It was this motivation that lead to the construction of very large water Cherenkov detectors deep inside mine tunnels where they would be shielded from most of the cosmic ray background and thus allowed to look for a signature showing the spontaneous generation of a shower from within the water indicating that a proton had decayed.

Such detectors, because they are so large and deep also make ideal atmospheric and possibly extraterrestrial neutrino detectors if they are big enough.

5.2a Irvine-Michigan-Brookhaven

The IMB experiment is a present collaboration of the University of California at Irvine, Boston University, Brookhaven National Laboratory, Cleveland State University, the University of Hawaii at Manoa, and Notre Dame University (the original "M", the University of Michigan at Ann Arbor, having dropped out). The experiment has been carried out in one form or another from late 1981 until the present. It is located at a depth of 600 meters (1570 meters water equivalent) in a salt mine run by the Morton-Thiokol company near Cleveland Ohio.

The detector consists of an approximately cubic tank (actually 22.5 m x 17 m x 18 m) of about 8,000 metric tons total of very pure water. The walls of the tank are reinforced concrete lined with two layers of plastic sheeting. All six sides are plastered with 2048 8 in. diameter hemispherical photomultiplier tubes. The outputs of all the photomultipliers are recorded for analysis later by an off line computer. For the PDK (Proton Decay) experiment, the final fiducial mass is reported out as 3,300 metric tons. The effective area for through going muons is ~ 400 m^2. This is an extremely flexible detector with triggering based upon a high number of photomultiplier tubes being involved within a coincidence time window. With so many tubes involved, the angular resolution of the instrument is < 5^0. The relative time of incidence information is carefully preserved so that track direction is known. Additionally, the presence of a muon in the final state of a low energy contained event, allows one to distinguish between electron neutrino and muon neutrino events. Thus, while waiting for the signature of a proton decaying within the fiducial volume of the detector, one can study thoroughly the through going muons and their angular disposition. A universal time (WWV) clock is used to date time stamp all recorded events.

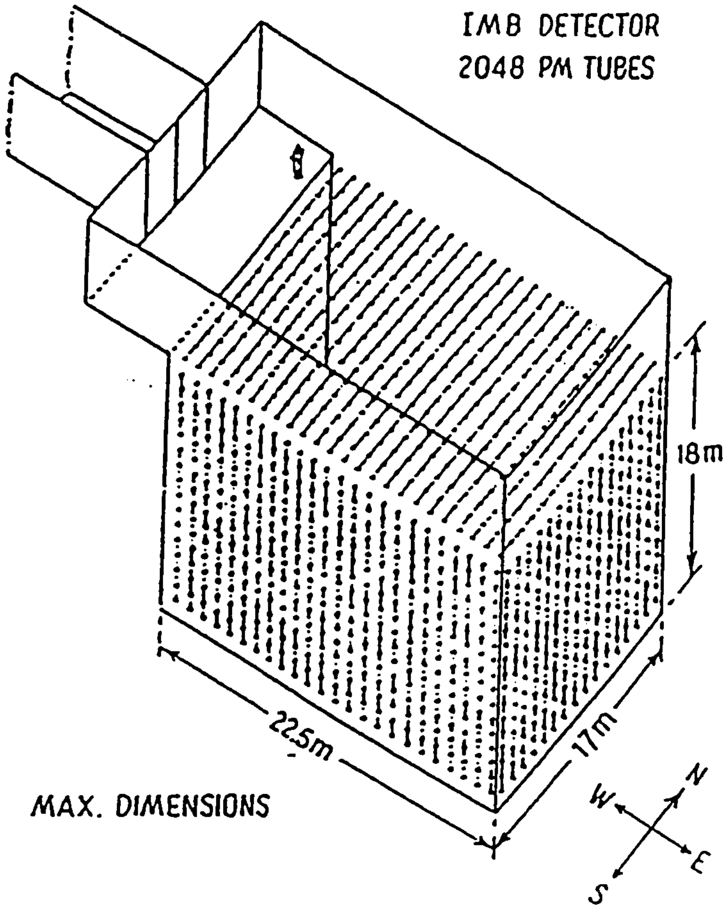


Figure 5.14 Cross section diagram of the IMB water Cherenkov detector

In fact, such a flexible detector has resulted in quite a number of different studies to include such things as proton decay, cosmic ray muon phenomena, searches for new types of interactions, neutrino bursts, studies of neutrino oscillations in > 300 MeV atmospheric neutrino fluxes, dark matter candidates, and the search for extraterrestrial neutrino sources.

5.2b Kamiokande

Similar to the larger IMB detector, a largely Japanese collaboration of the University of Tokyo, KEK, Tokai University, Kobe University, Niigata University, Osaka University plus the University of Pennsylvania, have constructed a water Cherenkov proton decay detector in the Kamioka mine 300 Km west of Tokyo. This is an active lead and zinc mine with a mountain rock cover of 2400 - 2700 meters water equivalent. The experiment began in 1982 and is on going.

The detector consists of a cylindrical steel tank with a volume of 3420 m^3 containing a fiducial volume of 1000 m^3. It is lined with 1000, specially designed, 20 in diameter photomultiplier tubes where 20 % of the external surface is covered by photocathode rather than just a few per cent. The basic advantage of this approach is that it improves the photoelectron statistics giving good energy resolution and allowing clear identification of pure electromagnetic decay modes so that different theories based upon the standard model may be put to direct test.

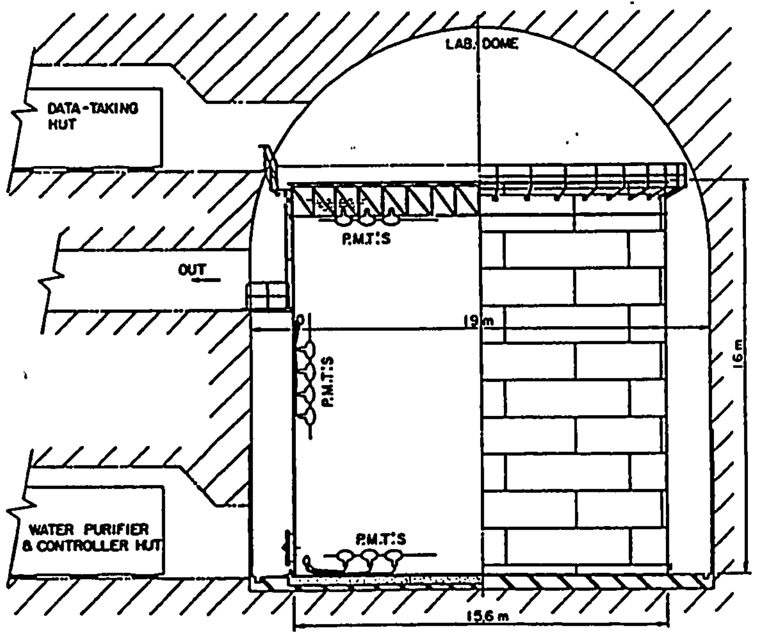


Figure 5.15 Cross-section diagram of the Kamioka mine experiment[16]

The pay off for all of this work after two decades of constant effort at greater precision in angular resolution and track reconstruction came in the early spring of 1987. A supernova, SN1987a, was reported by

[16] Grant 1982

observational astronomers to have occured within the province of our own galaxy, in the Large Magellanic Cloud. This was the first such supernova in the area of our galaxy in 400 years. After some time, the experimenters at Kamioka found an unambiguous neutrino burst around the estimated time of the supernova but could only pin down the absolute time to within six minutes or so having had problems with their clock. Taking this lead, the IMB group (actually the Hawaii branch of it along with Irvine) made low cuts in energy (around 25 MeV) compared to their usual threshold and also found the neutrino burst. This confirmed the first extraterrestrial neutrino source, confirmed the basic prediction that supernovas are driven by neutrinos, and accomplished the world's first clock calibration by a neutrino signal giving birth to the new observational science of neutrino astronomy!

These detectors, with their vast arrays of photomultipliers deployed in a large water volume, are the obvious predecessors to a DUMAND type array which uses much of the same kind of technology in about 4,500 meters of ocean water. Such a detector is, at least in principle, almost infinitely extensible, not being limited to the confines of a mine shaft. In contrast, the cost per m^3 of excavation is on the order of \$10 or so and such excavation is not practical with dimensions > ~30 m or so due to rock capability to resist collapse. The way is now clearly paved to making large deep ocean neutrino observatories such as DUMAND.

CHAPTER 6 -- Instrument Building Part I -- The String Communications System

6.1 Introduction

This chapter is the first in a series of five successive chapters which will document in detail those sections of the String Bottom Controller (SBC) for which I was directly responsible for in terms of their design, construction, testing, and systems integration. This chapter specifically deals with the "order wire" or slow speed command and control communications network linking up the various "smart" instruments found on the string.

6.2 Design Considerations

As was described above[1], the SPS consisted of a series of "smart" instruments strung out over a harness over 30 m long. Specifically, there were seven PMT's spaced 5.1 m apart, two calibration modules interleaved between the second and third and the fifth and sixth PMT's, as well as an environmental module all housed inside 17 in. (outside diameter) Pyrex glass housings isolating the equipment from the close to 400 atm. pressures found at DUMAND depths. Additionally, the environmental module was connected to a pair of hydrophones and an ocean current meter. All of these instruments were "smart" in the sense that they contained a microcontroller which could be directed externally to change switch settings and thus reprogram the instrument to meet changing circumstances.

[1] see Chapter 4

Not very long into the early design stages of the SPS project, it was recognized by the group that a command and control network would have to be developed so that a dynamic reconfiguring of the instruments could be done. The ocean is a very harsh environment. There are crushing depths which must be taken into consideration. It is electrically conductive and so wire based systems could easily be shorted out. It is corrosive, so that in the long run some exposed surfaces can literally vanish. Additionally, a combination of all of these factors can result in penetrating salt water making contact with different metals in different places, resulting through the different inherent electronegativities of the metals, in a "battery" being created causing unwanted currents that could play havoc with electronics.

With this all in mind, reliability became my overriding concern in designing a communications command and control system for the SPS. It would have to satisfy the twin requirements of being both simple (and thus with less things to go wrong, with less components to fail) and defensive in nature in that the attempt would be made from the start to anticipate some of the worst case problems and to design specifically around these.

An additional requirement, suggested in conference with the DUMAND collaboration, was to design the system in such a way that should it be considered meaningful to do so in the future, that the system would have the ability to not only exchange commands and data but also the ability to download a new controller program. The idea was to roughly have the same kinds of abilities that the Voyager space probes have had in that, as one's understanding of the environment evolved or as hardware tended to wear down but not completely die, one could still play games in controlling the system.

Finally, another consideration was dynamic range. As we shall see, it is possible to design a simple system in that it has few components that may well be able to work in the laboratory but still be self limited in its dynamic range. This is a potential disaster in that the harsh ocean environment is not as forgiving as the laboratory and once the system is deployed, there is little that can be done should for unforeseen reasons the overall communications channel begin to degrade. Thus, I considered it very important to have as wide a dynamic range as possible on the channel to hopefully overcome such bandwidth degradation.

6.3 Some Early Ideas

In this section, I will briefly review some of the early ideas that were floated around and point out what was wrong with them. Having considered such ideas is a useful exercise because it helps to focus more clearly upon an eloquent solution. The need for so much greater consideration is reliability. In the whole SPS instrument, there were three areas of extreme reliability. One was the fiberoptic cable, for without it mechanically, one would lose the instrument, without it optically, there would be no data. Another was the SBC instrument housing, which we tested time and time again in the ocean for if it did not protect the SBC electronics, again the experiment would not happen. Finally, there was the string command and control system. One could easily afford the loss of an optical module or a calibration module or even the environmental module but if one can not communicate with the modules then again there would be no experiment.

One of the earliest ideas was to use the seawater as a return. While this had the advantage that it required a virtual minimum in terms of communications medium (a single wire to complete the circuit), it was easily dismissed because should there be a pinhole leak in the remaining wire, all communications would be lost as both sides of the communications circuit felt the same (seawater ground) potential.

The next idea was to use even less wires. Rather than have a separate communications medium, why not use the power distribution system and superimpose the signal on the power lines? This was precisely what was going to be done on the 2 KA power line from the ship to the SBC so why not just do the same in all the string modules? This was not a good idea for several reasons. First, in order to drop and insert the signal at a given optical module, there would be the need for a very sensitive modem system which would strip away the signal from the power. Second, the system would have to be made foolproof with respect to power transients so a lot of error detection would have to be built into the system at different levels to ensure that a command got through and that a response was what was intended. Third, any such drop and insert scheme would require some kind of an isolation component (such as a transformer) which would result in a voltage drop of probably a couple of volts. Since it was decided that the different string modules would be fed 48 VDC and with ten such instruments each possibly loosing a volt or two, the overall line voltage would be too badly degraded to be of any use. Forth, such an arrangement would be very sensitive to the number of modules on line and there were real doubts about just how wide the dynamic range for such a system would be. Fifth, to do all of this would require a real increase in the complexity of the electronics (called a DAA for "Data Access Arrangement") thus decreasing the system reliability. Further, the increase in demand for circuit board space was particularly difficult to justify in the case of the optical modules where all of the electronics had to be built in the small area around the neck of the PMT.

From the above arguments, it was pretty obvious that a separate pair of wires along with another two wire penetrater through the protective housing should be supplied solely for the purposes of communications. This was a simple solution requiring no additional overhead in terms of board space nor error detection and correction programming. As such, it was intrinsically much more reliable. One scheme proposed the simple transmission of a standard 5 VDC serial signal with the different modules all wire "ored" together. This had the advantage of being simpler than the other schemes and in that sense more reliable. There were several problems with this. First, the probability of induced voltage spikes in a wire pair over 30 m long, especially when one of the instruments in the system is switched on or off, is not infinitesimal. Second, each TTL driver would be having to drive the entire length of wire (with its distributed capacitance cutting back on how fast a level may be shifted), a not inconsiderable load. Thus, a lot of work would need to be done to be sure that TTL logic level crossings are within specification. Say a command was sent from the SBC to the nearest optical module and to the farthest optical module, there would be a great difference in the voltage levels of a logic "high" (nominally

> 3.5 V) due to the difference in resistive loading and possibly a great difference in the time domain positioning of the logic crossings due to the difference in capacitive loading depending upon the baud rate selected. This may work, but the dynamic range of the system is limited to a very narrow linear region and so very susceptible to simple system degradation and voltage sensitive noise.

To overcome the problems of limited dynamic range, I decided to use a commercially available 300 baud modem chip. The chip selected was the Texas Instruments TMS99532. The principle of such communications systems is frequency shift keying (FSK[2]) which is a variation of frequency modulation (FM). In standard FM, one takes a whole range of analog voltages and associates a frequency shift (about a central frequency called the "carrier") with each possible voltage. In FSK, one takes two digital voltages, each one representing either a logic "high" or logic "low" and associates a unique frequency with each one. The TMS99532 was chosen because at the time of selection (the Spring of 1984) it was the only such chip readily available. It is an implementation of the Bell 103 telecommunications standard which supplies two communications channels (for the possibility of bidirectional or full duplex communications).

[2] Taub and Schilling, pp. 381-382

Table 6.1

Bell 103 (300 baud) standard frequencies[3,4]

communications mode:	frequency representing data logic level:	
	logic 1	logic 0
originate	1270 hz	1070 hz
answer	2225 hz	2025 hz

These frequencies, are all faster than twice the specified baud rate (which at 300 baud would give a Nyquist limited frequency of 2 x 300 = 600 hz) ensuring 100% sampling of the information. The main point was that 300 baud was "fast enough" for the task at hand and one did not have to go through the exercise of developing a modem from scratch which would take up a lot of time, money, and printed circuit board space.

As implied by the use of FSK, the modem contains oscillators and bandpass filters at fixed frequencies. Thus, communications is neither voltage level dependent nor is it greatly affected by capacitive loading which may delay transitions in an unequal way because with a pure sine wave, at worst there will only be a phase shift which will have no affect on the receiving bandpass filter. What this means, is that the dynamic range of the communications system is much larger than the TTL system. The specifications for this chip claim a typical analog receive voltage into the chip of 0.700 V p-p (volts peak-to-peak) (or -9.9 dBm)[5] with a minimum analog receive voltage into the chip of 25 mV p-p (or -38.9 dBm) for a minimum dynamic range of 29.0 dB! Additionally, since the transmit and

[3] TMS99532 Application Report, p. 2-3

[4] See Jennings, p. 46 for the similar but different V.21 European 300 baud standard.

[5] TMS99532 Application Report, p. 2-5

receive channels are all on different frequencies, it is possible to simultaneously use the same communications medium (wire) for all channels. Something quite impossible with the TTL approach.

Thus, the basic design now became one of a two-wire, common mode or party line 300 baud FSK modem based system. Such modem chips are originally designed to be used in point-to-point communications. To implement this, the SBC would have to have 10 separate communications ports making the SBC even more clumsy and, with more penetraters, more vulnerable to failure. Thus, it was decided to design a system in which the SBC had only one communications port making the string, if anything, more like point-to-many. Obviously some additional thought was necessary.

6.4 Problems Identified

This section will describe the worst case communications problems that were anticipated in the design and the strategies taken to prevent them. The details of implementation of some of these strategies will be found later on in other sections. In general, to make the system highly reliable, it was decided to try and take a defensive posture by identifying them and designing around them.

The first problem considered is a very common one in many communications systems. It is called the **screamer**. Whenever one has a communications system in which there is a shared communications channel among three or more nodes, then the possibility exists that one node may go into a "transmit always" (or intermittent "transmit almost always") mode preventing any other node from being able to transmit information. This is very serious, for while the mechanical integrity of the communications channel remains, the channel is now made useless and communications is lost.

I decided on several things to try and control the screamer. First, I designed a microcontroller based switch yard[6] as the main part of the SBC power distribution system. The idea was that individual power drops were being made to the different modules on the string from the central 48 VDC power supply in the SBC. Each power drop was made through a mechanical relay on the positive side of the current loop. This was essential for the orderly control of power to the individual modules. The basic idea was that the system as a whole, upon powering up, would be in a mode in which individual power drops to the different modules would all be off. Then, on command, each module, in turn, would be turned on. Among other things, this gave some control over communications on the string by allowing the operator to selectively turn on and off modules until the screamer was shut down and string communications could be restored.

[6] discussed in detail in Chapter 6

Another approach to handling the screamer came in the form of a screamer killer circuit designed to automatically shut off a screamer for most of the time. The modem chip has a squelch control, in other words an output disable line, which can be used to disconnect any analog signal output from the string interface circuit (known as the Data Access Arrangement or DAA - this is discussed in detail in the next section). At 300 baud, a ten bit long character (one start bit, eight data bits, and one stop bit) would allow 30 characters to pass by per second. It was visualized that no message stream would ever be as long as 150 characters or five seconds in duration. With ten instruments on the string, this means that in ten times five or 50 seconds one could completely poll around the string. The screamer is effectively disabled if the transmitter is allowed no more than its maximum transmit window time on the reply channel and then squelched for another 45 seconds for a total of only one access per poll time. If it transmits at that time, it is merely garbage that can be ignored without it interfering with the other modules. As such, the screamer killer circuit is a series of timers, completely independent of the microcontroller (should that be the most likely reason the module goes into a screamer mode) which detect the first transition of a message string and time everything out controlling transmission through the squelch control line of the modem chip.

The second problem is the exact reverse of the first. The communications system could just as catastrophically fail should one node continuously demand information from one or more other nodes. In a sense, the system would be entirely tied down by an information **black hole** constantly sucking up information for no purpose.

This problem was precluded by giving network control status to the SBC side of the string with all the other modules being treated equally. In such an arrangement, the individual modules on the string are never allowed to transmit unless they receive a command demanding that they do so from the SBC. This "reply only on command" approach eliminates all the string modules from ever being able to demand information except the

SBC and should the SBC side fail anyway there would still be no communication.

Finally, there is the problem I have called the **great flood**. As has been mentioned before, the deep ocean is a potentially very harsh environment. The probability of one of the modules failing due to a catastrophic implosion (or even a clam shell sort of gulping of seawater as the two hemispheres of a protective housing accidentally leak while going through the air-water interface) is probably higher than one might like to think. It would be catastrophic should the communication lines see the same potential due to a seawater short in a single module. A physically simple, yet reliable, method of protecting the communications system from such a flood was needed. After many false starts, a physically very simple solution was found. All of the communications lines were isolated from the sphere interior by being coupled between the interior circuits and the electrical penetrater through a series resistor. These resistors were in turn mounted directly onto the interior side of the electrical penetrater and then physically "potted off" or sealed from the interior. Thus, in the worst case, should there be a seawater short circuit, the separate communications lines would not see the same potential. In time, the exposed end of the resistor would corrode and as such increase in effective resistance loading down the system even less (since the different modules and thus their resistive paths are in parallel to one another).

6.5 The Modem and the Data Access Arrangement

In order to inject a strong (transmitting) frequency onto a common channel and not suffer some kind of local feed back effect in the transmitting modem through frequency spill over in its receiver section, some means is needed to make the receiver deaf to the local transmission. Typically, in broadcasting situations, the local receiver is "keyed" or switched out of the circuit just at transmission time. In the case of old fashioned ionospheric bounce high frequency stations, the receiving station was located perhaps twenty miles away from the transmitting station! In any event, communication was simplex, namely information could only be transmitted in one direction at a time. However, in our case, with a little reflection, a DAA circuit was designed which overcame this problem permitting full duplex communications, if so desired, and absolutely no need for the communications system to have a means of switching off between transmitters.

The DAA circuit shown in Figure 6.1 below consists of two LM324 operational amplifiers arranged in a sort of a bridge network so that the receiver (node 4) does not see the transmitter (node 1) during transmission. By itself, the first amplifier acts as a transmission amplifier arranged as a unity gain buffer driving a 390 ohm load (as the string at node 3 will appear to it) which is also the "potted resistor" on the transmission side installed to preclude the great flood problem mentioned before. There are two pathways to the receiver. The first, is from the string through a unity gain buffer identical to that used in the transmitter. The second, comes straight from the transmitter through 390 ohm and 100 K series resistors exactly matching the current (due to local transmission) directly from the output of the transmitter amplifier to the input of the receiver amplifier. Since the transmitter amplifier is an "inverter", the phase of these two transmission signals theoretically should exactly cancel at the input to the receiver amplifier. Of course, the phase inversion going through all of the components is not exactly 180^o but it is pretty close, especially if the unity gain buffer of the transmission amplifier is properly adjusted so the feedback signal can be smaller than any expected receiver signal from the string and properly adjusting the receiver amplifier gain can separate the two. Note that the 100 K resistor in series with the receiver amplifier is the "potted resistor" on the receiver side of the communications line.

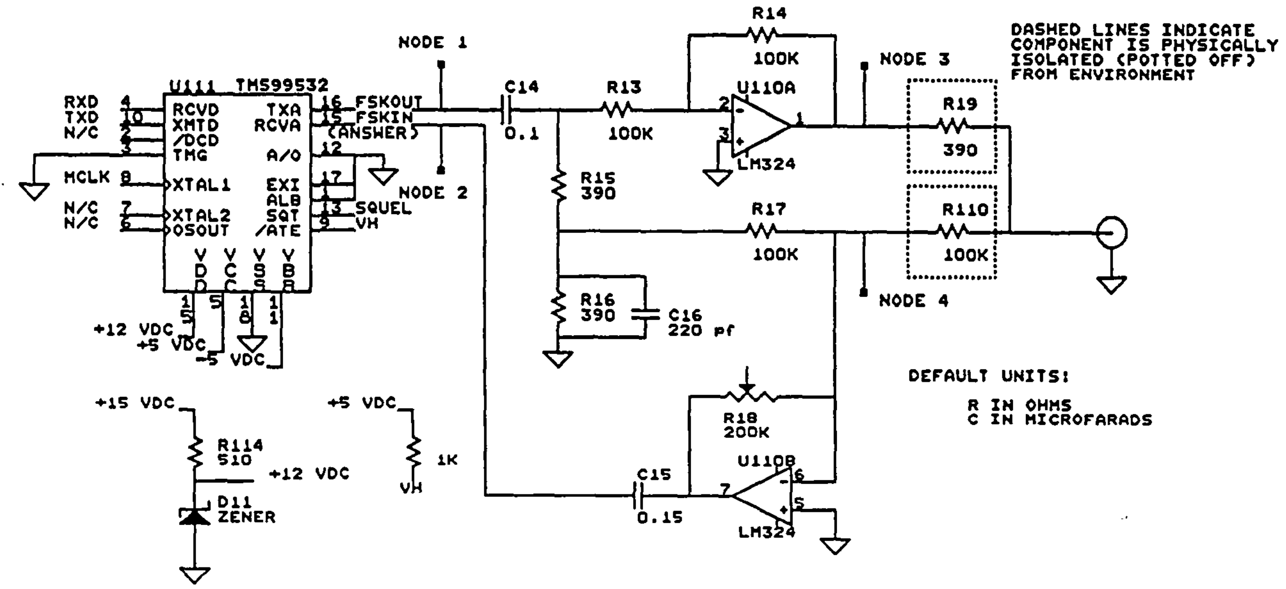


Figure 6.1 "Nulled" or two wire based DAA circuit.

This arrangement is precisely what was used all the way through the Ersatz SBC cruise of November 1985. Marginal transmission errors kept occurring in some of the modules and it was decided to take another look at the circuit. Most particularly, as was noted above, each one of these circuits had to be carefully "nulled out" in order to minimize the transmitter feedback. This could be adjusted easily enough in the laboratory with the protective Pyrex housings off. However, the real situation was one in which the housing is closed, with additional electronic noise due to the many switching DC to DC converter power supplies as well as other oscillators all in the same small enclosure. Additionally, the operating temperature of the modules in the deep sea was closer to 4 ^{0}C rather than room temperature. As such, the settings made in the laboratory were probably not optimal for actual deployment.

Looking around at what was available, a solution to this problem became obvious. Originally, the return path of the communications channel was a separate (ground) line isolated from the power ground. This was not really necessary since (let us say there was indeed a flood), if the ground were suddenly forced to be at the same potential as the seawater while the communications line remained isolated from it through the potted resistors, no harm would be done. In this scheme, all transmission and reception occurred over the exact same common communications line with the channels separated only by frequency. Now, instead of having a separate return line for the communications circuitry, the common power return was used. This was possible because the power distribution logic in the SBC only switched the hot side of the power, not the ground side. Thus, an additional communications line was freed up. As such, the solution was to make one line from the SBC communications port to be a "transmit only" line connected with only the receiver sections of all of the string modules and to make the other line from the SBC communications port to be a "receive only" line connected with only the transmitter sections of all of the string modules thus completely isolating the receivers from their transmitters. This change is shown schematically in Figure 6.2 below and resulted in a much reduced incidence in communications errors.

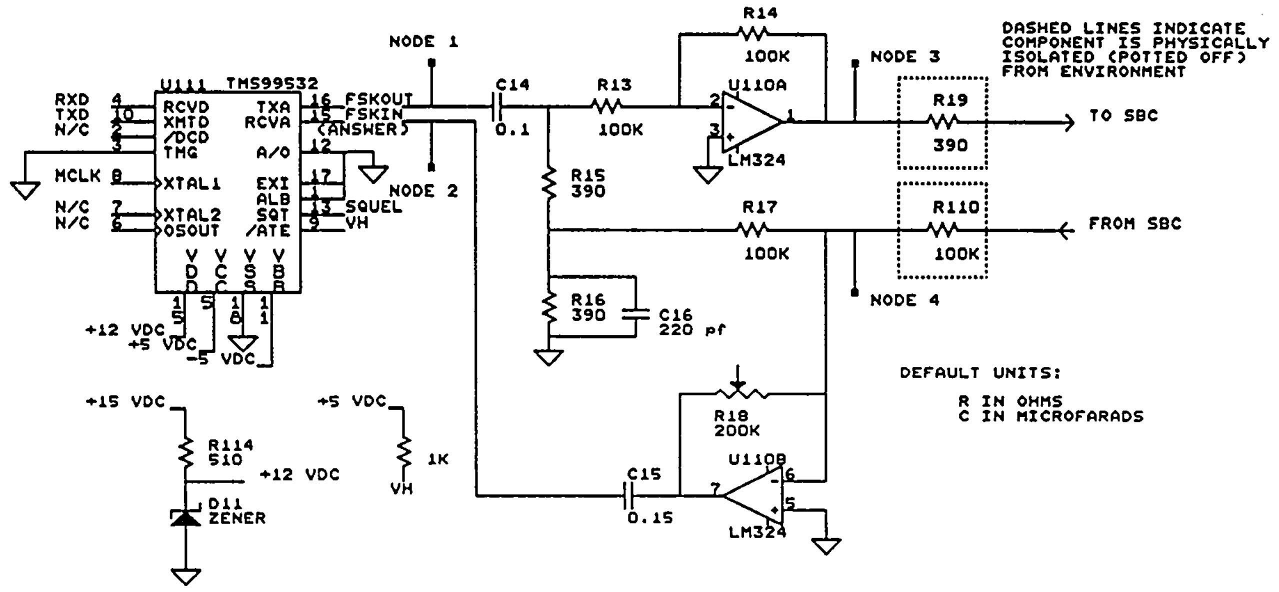


Figure 6.2 Three wire based DAA circuit.

6.6 The Screamer Killer Circuit

The basic design philosophy of the screamer killer circuit is to have an operational definition of a screamer which can be used to detect such a situation completely independent of the microcontroller since it is the failure of the microcontroller itself that would probably constitute a worst case scenario for the screamer. Once this condition was detected, then the circuit should squelch all output from the modem chip long enough for all the other modules to be polled once. Since a screamer is defined as being any continuous transmission exceeding five seconds, then this means that the maximum total poll time for all ten of the string modules is 50 seconds.

The procedure used to define a screamer is to trigger two monostable timers upon the beginning of a transmission by having their (low going triggered) inputs look at the transmitter data line TXD. The first monostable called the "packet repetition" timer is set with a time constant five seconds long. Should the monostable time out and transmission is still in progress, then a screamer condition exists. To detect this condition, it is necessary to sample whether or not the modem is still transmitting. This is done through the use of the second monostable or "transmission active" timer which is set to 1/20 th of a second or about the length of time it takes to transmit a little less than one byte at 300 baud. As such, it can be used to sample on a byte by byte basis as to whether or not a byte is being transmitted at the time. As a monostable, it is retriggered every time a new low going transition occurs as soon as it times out. A small capacitor is added to the timer output as a choke so that upon timing out the line will not immediately return to ground. Since, generally, this timer will immediately be retriggered by another transition, this prevents glitches from getting through every time the timer is retriggered. Once transmission of a packet really is complete, the monostable will time out and not be retriggered so a low level output at this point indicates that transmission is not active.

If T1 and T2 are the respective outputs of the first and second timers, then the logic condition defining a screamer is given by (!A means "not" A)

$$S = T1 + !T2 \qquad (6.1)$$

where a low level indicates a screamer condition and a high level indicates that there is no screamer.

The low going transition on the S line which indicates a screamer condition is used to trigger a third monostable called the "poll timer" which has a time constant of around 45 to 50 seconds (one string poll time). The output of this timer is used to gate off the transmitter data line TXD. If a screamer never occurred, then this monostable does not get activated and the module is free to transmit again without having to wait out an entire polling cycle which is really useful should one wish to successively interrogate the same module (such as an almost continuous read out of PMT single's rates data, etc.).

The monostable timer used in this is based upon the famous Signetics Corporation NE555 timer[7] whose time constant (T in sec.) is adjusted by an external resistor (R in Ω) and capacitor (C in μf) given by the formula:

$$T = 1.1\,R\,C \qquad (6.2)$$

The actual version used is a dual CMOS based one designated the ICM7556 and manufactured by Intersil Corporation.

[7] Signetics Analog Timer Manual, p.2-3

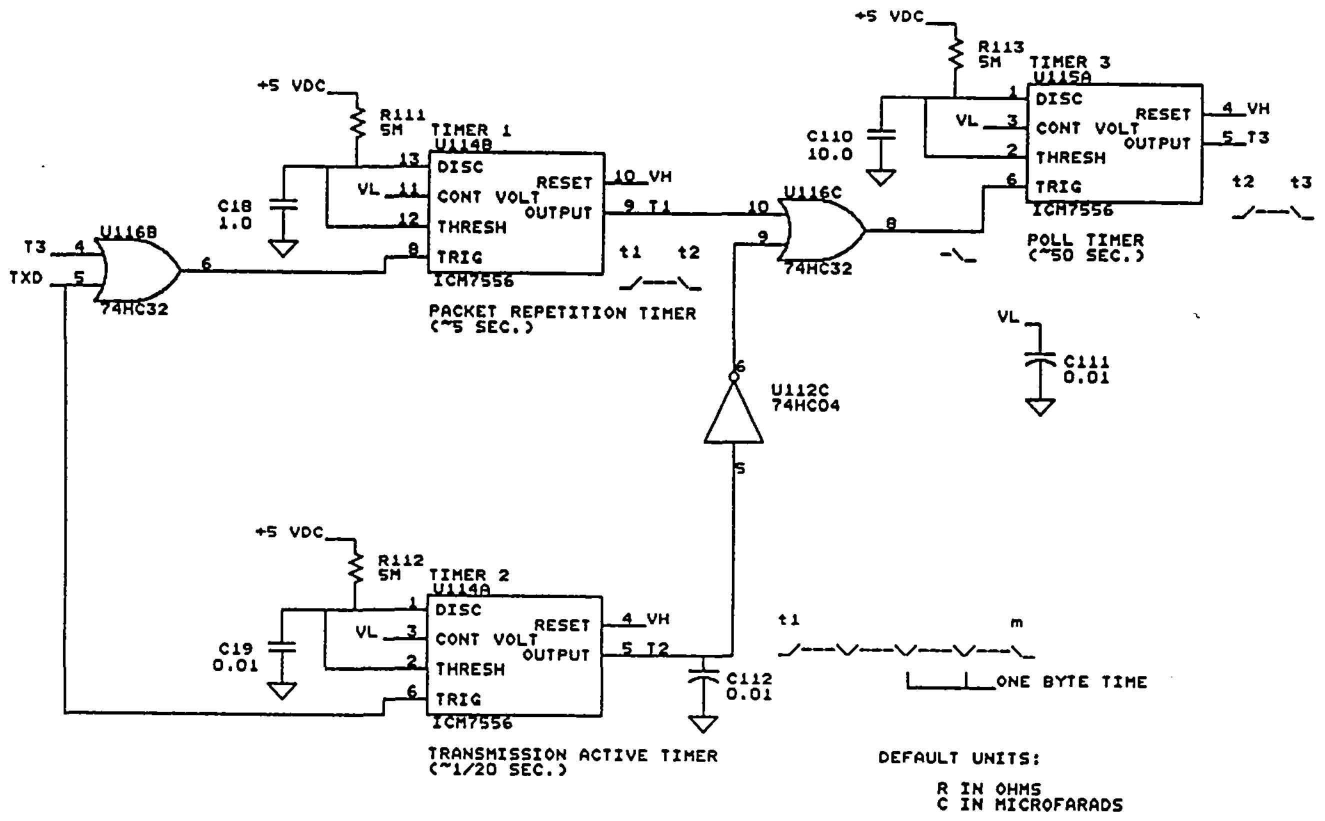


Figure 6.3 The microcontroller independent screamer killer circuit. Note the use of three asynchronous timers to detect and squelch off transmission.

6.7 Protocol

The message protocol is that part of the communications scheme whereby the very nature of the information stream itself is used to help control communications. It is somewhat akin to the grammar of a language. The protocol was chosen from the point of view of being simple and unambiguous to use.

First off, it was decided that the information stream would be represented only by a stream of printable ASCII characters. In this manner, it would be possible (maybe not desirable but possible) to control and monitor communications with a simple ASCII keyboard. Should the need arise in the future for downloading programs, then the program would have to be exchanged in some representation other than pure binary code such as a hexadecimal or octal representation.

Second, it was decided that all messages would be made up of message packets. In this way, it was possible to impose a maximum packet length per message unit and thus easily define a screamer as any transmission that exceeded this length. Long messages, such as a computer program, could be made up of multiple message packets. In practice, this was never done since for the SPS demonstration project, there was never a need to download any program changes but the structure for doing this within the communications system exists. Only short commands followed by short replies were ever needed.

The structure of a message packet was as follows:

```
$<dev><dev><cmd1><cmd2>[<data1><data2>]<CR>
```

where '$' is the dollar sign character in the ASCII character set used as a message packet[8] synchronization character indicating the beginning of a message packet. <dev><dev> indicates two identical characters containing a single digit hexadecimal number that was the unique address of a module on the string. <cmd1><cmd2> indicates a two character command to be performed which was followed optionally by two data bytes (indicated within the [ ] as <data1><data2>). Finally, the ASCII carriage return (<CR>) character was used to indicate the end of the message packet.

The structure of a reply message packet from a module to the SBC was even simpler being

$<cmd1><cmd2>[<data1><data2><data3><data4>]<CR>

since only one module was permitted to reply at a time, the interrogated device, the device replying was unambiguous and there was no need for it to identify itself.

[8] Roden, pp. 480-481

6.8 Summary

A practical local area network (LAN) for controlling a series of ten "smart" instruments mounted on a vertical string in the deep ocean was developed. The network has a network controller collocated with the instrumentation for channeling the data from the instruments to the surface (the SBC). It was felt that giving this job to the controller made sense since should the controller housing fail, the system would cease to function anyway and as such there was no significant increase in the probability of catastrophic communications failure. In turn, the simplicity and hence reliability of such a "reply only on command" network without any need for things such as "collision detection logic" when two nodes try to transmit at once was greatly increased.

Three distinct worse case scenarios, the screamer, the black hole, and the great flood, were identified and defensively designed around. The LAN is a 300 baud FSK (Bell 103) based system which originally was implemented as a two-wire one point to many point arrangement with good separation between the local transmitter and receiver circuits. Later, this was changed to a three-wire arrangement with excellent separation between the local transmitter and receiver circuits.

A simple, printable ASCII only, message packet protocol was chosen to so that communications could be easily monitored using standard ASCII based terminals and to make the detection of a screamer fairly easy.

CHAPTER 7 – Instrument Building Part II – The Design of Microcontroller Circuits for the Optical (Calibration) Modules

7.1 Introduction

This chapter is the second of a series of five on instrument building of the Short Prototype String. The main concern of this chapter is to document the work I did in designing devices based upon the Intel Corporation 8051 family of microcontrollers. This is not a complete documentation of all of the work done but rather a selective documentation of the hardware and software actually used in the deployment of the SPS during the October-November 1987 cruise off of O'ahu and Hawai'i. For the sake of rough completeness, I mention in passing that individual test programs were developed to exercise and debug each of the hardware driver modules. In turn, these were collected together, in some sense, and an overall ROM based test program was developed for testing the optical modules and as an early exercise in system integration of the optical module. Finally, in support of the November 1985 "Ersatz" SPS cruise off of O'ahu, a separate 8051 based hardware and software system was developed to control the Ersatz SBC which was an attempt (failed) to do photomultiplier tube coincidences in the deep ocean with an as yet incomplete SBC. The failures were due to mechanical problems associated with the great difficulties of working in the deep sea environment and were of great use to us later in learning where to concentrate our efforts.

Before I get into the details of the circuitry, it is important to explain the difference between a "microcontroller" and a "microprocessor". In a typical computer, one has both an address space and a data space. The address space is a series of (parallel) binary control lines which in combination make up a sequential set of unique binary values (addresses) which may be used to "address" memory cells or "ports" which in turn control external hardware that form a basis for communications with and control of the external environment. The data space is a similar set of binary control lines used to pass parallel data in and out of the computer's "processor" logic. The processor logic, in turn, consists of a series of internal registers which temporarily hold all sorts of information such as the current instruction being executed, the program pointer pointing to the current instruction, a stack pointer pointing to the (usually) external stack (which is a sort of "scratch pad" for information), the operands associated with the current instruction, the accumulator(s) which are registers within which the actual logical operations are performed, condition codes, etc. as well as the all important arithmetic logic unit (ALU) which is the logic that recognizes the instruction and carries out its execution.

The arrangement or "architecture" described above for a typical computer is what one finds in a microprocessor. A microcontroller differs from this in one principle feature of its architecture. A microcontroller has a sort of third space, a controller space, associated with it. In practice, what this means is that in addition to the address and data lines of the microcontroller, there are many additional lines which can be directly toggled on and off for the control of external hardware. The basic idea is that such an implementation eliminates or reduces the need for address decoding logic to decode dedicated addresses in the address space in order to control external hardware. In short, the required amount of board space to implement a given system should be less than that required by a microprocessor based system.

At the time of selection[1] a world search was made for commercially available microcontrollers. The decision to use a microcontroller rather than a microprocessor was made based upon one important constraint, namely that the optical modules have very little space[2] in them to house their electronics. The available space is restricted to the volume around the neck of the photomultiplier tube. In this space one must accommodate a

[1] summer 1983

[2] Note the limited space around the PMT neck in Figure 4.2.

vacuum gauge, the programmable high voltage supply for the photomultiplier tube, the pulse stretcher circuit, the fiber optics driver, the various analog sensors, the dark current isolation amplifier, the 300 baud command and control communications circuitry[3], and a whole array of 48 VDC to DC power supplies. In fact, it was several months and quite a number of different ideas before a two tiered printed circuit board arrangement was finally decided on to mount all of this hardware.

The world search turned up only a few commercially available microcontrollers. The hope was to select a microcontroller with sufficient hardware to accomplish the job at hand and ultimately be available in a CMOS version so that in large scale production circuits could be developed drawing less current. The first was the Motorola Corporation's 6803. The main advantage of this chip is that its instruction set is virtually identical to the 6800 microprocessor. This is a very powerful (for an 8 bit wide machine) instruction set with many addressing schemes for which at the time a lot of software already existed. Most particularly, it does not follow the Intel Corporation fashion of forcing all operations to be executed through the bottleneck of a single accumulator register[4]. In terms of its hardware, the MCM6803 has many nice features. These are summarized along with the similar features of the other possible microcontrollers below in Table 7.1.

The second microcontroller was the Intel 8048 family. This was an older microcontroller in the sense that Intel had just recently released a more advanced microcontroller, the Intel 8051 family, and thus it was never seriously considered. The Intel 8051 was selected based upon the comparison of the advertised hardware features as shown in the summary table where the designations 8748 and 8751 are the UV erasable ROM versions of the 8048 and 8051 microcontrollers respectively:

[3] See Chapter 5.

[4] See Chapter 8 for details.

Table 7.1

Hardware feature comparison of available microcontrollers[5]

Feature	MCM6803	8748	8751
Parallel I/O lines	13	27	32
On-board RAM (bytes)	128	64	128
16 bit timers,	1	1	2
Serial I/O ports	1	1	2
UV ROM (kilobytes)	0	1	4
Total addressable memory (kilobytes)	64	64	128

As such, at the time, there was little choice since there was no firm system description for the optical module nor would there be for some time so the microcontroller with the most built in hardware would be the logical selection. Unfortunately, the instruction set of the Intel 8051 microcontroller family leaves much to be desired[6].

The internal registers of the Intel 8051 are structured in such a way as to directly accommodate the hardware. Of real advantage to the problem of the optical module is the availability of two serial I/O ports. This permitted the establishment of full duplex communications[7] in the SPS order wire. Using the serial I/O ports meant the loss of one timer port to establish the baud rate. Still, one timer was left and, in the case of the optical modules, served the important function of timing out and reporting the photomultiplier singles rate as a crude measure of activity. A single 12 Mhz crystal provided timing for all the synchronous hardware of the

[5] Note, typically to do serial communications, one must dedicate a timer for establishing the baud rate.

[6] Chapter 8 is dedicated to precisely this problem.

[7] As was described in detail above in Chapter 5.

microcontroller. In truth, only about 13 parallel I/O lines (the number advertised on the Motorola MCM6803 chip) were really available for controlling hardware. This is due to the fact that the control lines of the 8051 family are not dedicated but rather can be selected for different functions. For example, the serial I/O ports and the timers are listed both by their function and as parallel I/O lines. Additionally, many of the serial I/O lines actually have to be dedicated as address and data lines for an external data memory. The advertised internal data memory of 256 bytes (not kilobytes) is actually much more limited since should one wish to implement a stack, 48 of the internal bytes is the largest stack possible plus 128 bytes are dedicated to the direct control of all of the on-board functions leaving a total usable internal data memory of only 80 bytes! As such it becomes necessary to implement an external data memory, because there is almost no room for program variables in a program as sophisticated as the optical module executive.

7.2 Optical Module Circuit Description

Before I dive directly into the circuit descriptions, it is worth while to consider briefly what the objectives were in designing the optical module and very briefly how these were attacked. After that, a detailed description of the optical module digital circuitry will follow.

The main objective for the optical module was to provide a programmable or "smart" photomultiplier which would be capable of detecting the faint and rapid Cherenkov flashes from passing muons in the surrounding seawater medium. At a minimum, there was the need to be able to adjust the gain of the photomultiplier tube. This is accomplished by varying the applied high voltage across the dynode chain. Thus, a programmable power supply would need to be selected which would adjust the output voltage based upon the different analog control voltages applied. To select a desired analog control voltage and thus a desired gain, an eight bit D to A (digital to analog) converter, the Analog Devices AD7528 was chosen. Similarly, a threshold control as the front end of the photomultiplier tube output pulse stretcher circuit needed to be controlled so that a signal above the ambient noise would be selected. One may not normally think of it, but in some cases the PMT is used more for its ability as a noise source than as a light detector. During World War II, the application of PMT's as a broadband white noise source[8] increased production from hundreds per year to thousands per month. Again, an AD7528 was selected for the job.

In the opposite direction, there was the need to be able to sample a collection of sensors. The DUMAND SPS optical module probably has more sensors than is actually needed but it was useful engineering wise in understanding and testing out the concepts incorporated in the SPS test. Altogether, allowance was made for up to 16 analog sensors to be

[8] By applying a non-modulated input light source and operating with high gain so that the amplifier photoelectric shot noise was "white". See (RCA) Photomultiplier Handbook, p. 5.

accommodated. This was accomplished through the use of the National Semiconductor ADC0816 which is an eight bit A to D (analog to digital) converter with a built in 16 channel multiplexer capable of sampling an addressed channel[9] in about 100 μs. Ultimately, 15 of these were assigned as shown in Table 7.2:

Table 7.2

Analog sensor channel assignments in the optical modules

Channel number	Analog signal	Assignment
0	ANA0	unassigned
1	ANA1	+5 VDC A
2	ANA2	+5 VDC B
3	ANA3	-5 VDC
4	ANA4	+15 VDC A
5	ANA5	+15 VDC B
6	ANA6	-15 VDC A
7	ANA7	-15 VDC B
8	ANA8	total current
9	ANA9	dark current
A	ANA10	temperature A
B	ANA11	temperature B
C	ANA12	temperature C
D	ANA13	accelerometer
E	ANA14	PMT high voltage
F	ANA15	discriminator threshold

As you can see from the table, seven of these sensors monitored the different DC to DC converter power supplies and two monitored system

[9] See specification sheet for the ADC0816 in the National Semiconductor Linear Handbook.

currents. Three temperature sensors were mounted in different regions of the optical module to get some idea as to thermal distribution within the limited housing. One channel was used for an accelerometer to get some idea as to the accelerations the module was feeling since we were concerned about stimulated bioluminescence. Finally, two channels were dedicated to directly monitoring the actual values that the two control signals, namely the high voltage and discriminator threshold, settled at. This was considered very important since we did not want to get into the situation of thinking that by issuing a command that the intended value was absolutely the actual value. This gave us real feedback. It would have been possible to develop an algorithm to take this information and have the microcontroller compare it to the intended value and then automatically modify the control value output until, in a closed loop, the system settled down into precisely the value desired. Upon reflection, it was desired not to do this because it was potentially very dangerous should the system go into some sort of hunting mode by always slightly missing the mark. The result could be a system that would continuously try and fail to reach the exact desired value and thus oscillate or "hunt" about the decided value and so guarantee a variable value rather than some settled constant.

Finally, there was the objective of providing a communications interface. This was cleanly done through the use of the Texas Instruments TMS99532 300 baud modem chip along with a specially designed DAA (Data Access Arrangement) circuit and a screamer killer circuit.[10] Basically, the digital side of the modem chip was directly connected into the microcontroller's two serial I/O ports (one receive only, one send only). Timing was derived from a common clock circuit used to provide timing pulses to both the modem and the A to D converter.

The best place to start the description of the digital circuitry is with its central controller, namely the Intel 8751 microcontroller. This is documented in Figure 7.1 below:

[10] This was the subject of Chapter 5.

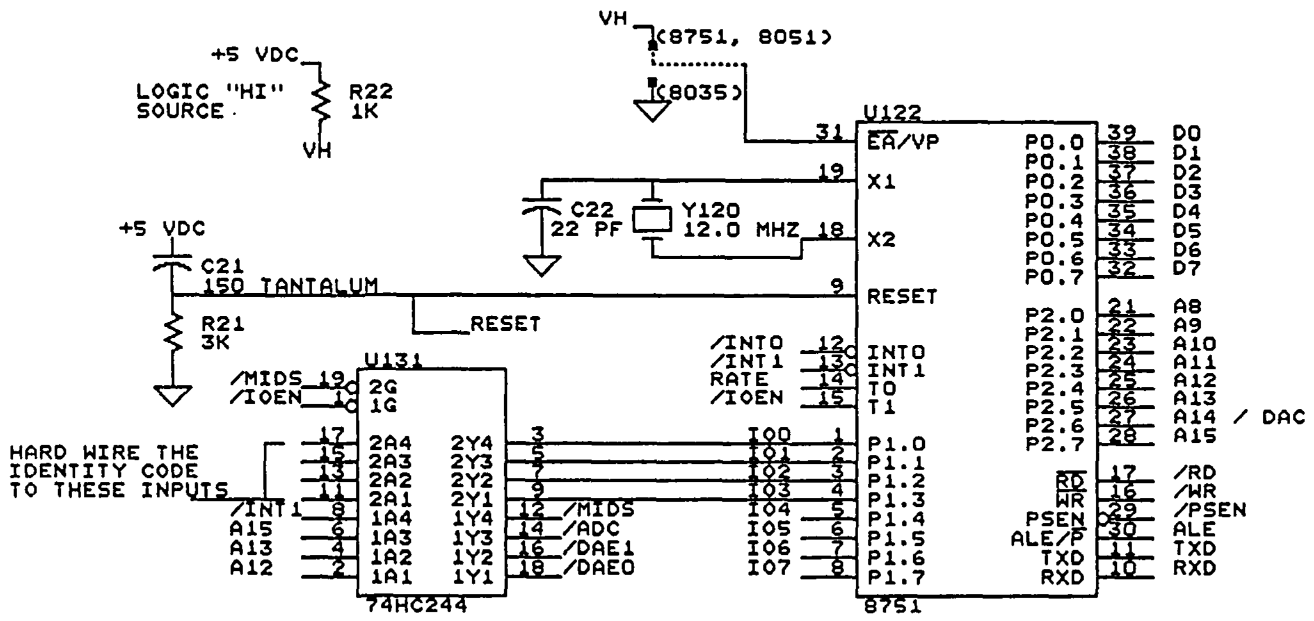


Figure 7.1 The Intel 8751 based microcontroller circuit.

The 8751 microcontroller (U122) is a 40 pin chip requiring a single +5 VDC supply. Timing is generated by directly connecting a 12.0 Mhz crystal (Y120) across the two pins XTAL1 and XTAL2 along with a decoupling capacitor C22. Internally, the main control cycle timing of 1 Mhz is derived through a divide by 12 circuit. This same clock is used for the two 16 bit timers (T0 and T1), one of which is ultimately used to generate the baud rates for the two serial I/O ports (RXD and TXD). Two levels of interrupts are available (INT0 and INT1) as well as the write (WR) and read (RD) memory control pulses. These last eight lines constitute parallel I/O port 3 of the 8751.

System start up is accomplished with the resistor and capacitor combination made up of C21 (a 150 pf tantalum) and R21 (3 kΩ) tied directly into the RESET line. Upon power up, C21 feels the +5 VDC supply and is charged thus pulling the RESET line high. In time, R21 bleeds off C21 so that the RESET line goes low. The power on reset requirements for the 8051 microcontroller are such that the RESET line should be held high for at least one ms after the power supply stabilizes to allow the oscillator to stabilize. The microcontroller remains dormant until the line goes low. The other lines of the microcontroller assume their initialization states within 24 oscillator periods. Upon the RESET line going low, a sequence is initiated which takes about 12 oscillator periods before ALE is generated (permitting dialog with the memory) and normal operation begins with the program counter pointing to the first instruction at absolute program memory location OOOOH.[11] The EA (External Address NOT) line is used to tell the microcontroller whether it is an 8751 or 8051 version (logic high condition) of the microcontroller which has some on-board ROM with a program in it or not as is the case for the 8035 version (logic low condition). This way, when the microcontroller starts up, it looks in the right place to find its controlling program. The symbol VH stands for a common logical "high" line that can be used in all cases where a digital (+5 VDC) high is needed. It is formed by tying a 1 kΩ resistor into the +5 VDC supply. In this way, less

[11] MCS-51 User's Manual, p.2-28.

current is drawn by the chips than if they were directly tied in. This also means that should an over voltage occur, these inputs which are more sensitive than the power inputs, are afforded a degree of protection through current limiting.

The eight parallel I/O lines comprising port 0 are dedicated as the data transfer lines D0 through D7 inclusive. Actually, these lines are multiplexed in time so that they also act as the lower eight address selection lines for external memory. The ALE (Address Latch Enable) output of the microcontroller is the strobe signal for differentiating these two roles. The eight parallel I/O lines comprising port 1 are dedicated as the principle I/O lines for transferring information from the microcontroller to the different D to A channels and from the different A to D channels to the microcontroller. These are labeled IO0 through IO7 inclusive.

The eight parallel I/O lines of port 2 basically are divided into two different sections. The lines labeled A8 through A11 are used to address the ADC0816 (U130) 16 channel A to D converter picking out one of 16 possible channels. Lines A12, A13, and A15 are used (among other things) along with the interrupt line INT1 as inputs to one half of U131 (74HC244 non-inverting bi-quad tri-state buffer) to generate the chip enable lines DAE0, DAE1, ADC, and MIDS. These are separated in time from other uses of some of these lines by strobing with the timer output T1 (labeled IOEN for I/O Enable). Table 7.3 documents the use of these different chip enable signals:

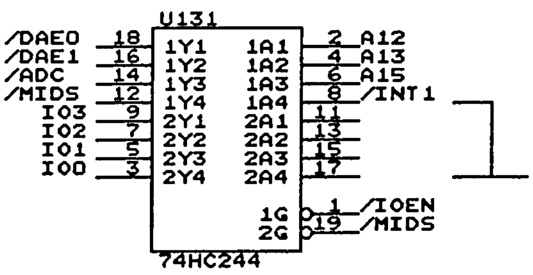
Table 7.3

Assignment of the chip enable lines in the optical module.

Signal	Assignment
<u>DAE0</u>	D to A Enable 0 (High voltage control)
<u>DAE1</u>	D to A Enable 1 (Threshold discriminator control)
<u>ADC</u>	Multi-channel A to D (16 channels)
<u>MIDS</u>	Module Identification System

Finally, in terms of the direct microcontroller associated circuitry, there is the module identification system. The need existed to be able to assign a unique identifier to each of the modules on the string. One possibility was to assign this identification in the ROM code. In principle, this would be easy to do but in practice it would be a disaster. The danger lay in accidentally assigning the wrong identity to a given module. If the problem was one of a simple exchange, then commands intended for one module would go to another one and, until the mistaken identity was corrected, both tubes would essentially be limited to their default values and feedback from them (on the 300 baud order wire) would be useless. A variation on this, which would be harder to correct, would be to accidentally assign the same identifier to two tubes. Then you would have a real data collision from both tubes trying to reply to a single command simultaneously. Ultimately, at least one of the modules would have to be opened up to correct the problem. All of this boils down to the problem that one would have to generate multiple versions of the same ROM with only one byte being different and the need to have one hundred percent backup should the microcontroller give out in a module since a universal part would no longer exist.

Out of this understanding, a hardware module identification system was developed, documented below in figure 7.2, which would allow the optical module assembler to identify the module by visual inspection should if be necessary. Basically, a series of four input lines were soldered on the board to either a logic high or logic low state and tied into the remaining half of U131 (74HC244 non-inverting bi-quad tri-state buffer). At the time the program wants to establish the identity of the module, the MIDS signal is asserted as a strobe and the output of the buffer is fed as input lines IO0 through IO3 inclusive into the microcontroller.



MODULE IDENTIFICATION INPUTS TIED "HI" (VH) OR "LO" (GND) DEPENDING UPON UNIQUE POSITION OF MODULE ON STRING

MODULE NUMBER	IOn BIT PATTERN 4	3	2	1
1	0	0	0	1
2	0	0	1	0
.				
.				
F	1	1	1	1

Figure 7.2 The optical module identification system.

An external memory circuit was provided from the beginning for the purposes of easing the problem of system development and later found to be absolutely essential. The Intel 8051 family is weird[12] in comparison to other eight bit microcontrollers and micro-processors. One of these areas of "weirdness" is in its memory design. As mentioned above, it is advertised as having a 128 kilobyte addressable memory space. In reality, it actually has two 64 kilobyte memory spaces which have absolutely nothing to do with each other. The two separate memory spaces are called the "code memory" and the "data memory". This separation of code from data is known as the Harvard architecture. It is in contrast to the more common total mixing or interleaving of code and data in the same memory space known as the Von Neumann architecture.[13] In practice, what this meant for optical module development is that code could never be down loaded and thus one of the original intentions with the optical module, to update the program should better algorithms be developed in the future, would not be possible with this microcontroller.

I said that the implementation of an external memory became essential because it did not take too long in program development before more than 79 internal data bytes was achieved just to handle program variables. Figure 7.3 documents the external memory circuit:

[12] This is discussed in detail in Chapter 8.

[13] Named after Johnny Von Neumann who first thought of the idea. He wanted to do artificial intelligence where a given piece of information in the common memory might be interpreted alternately as code and as data so that through some feedback mechanism ("learning") the code (data) is updated through inter-action with the environment.

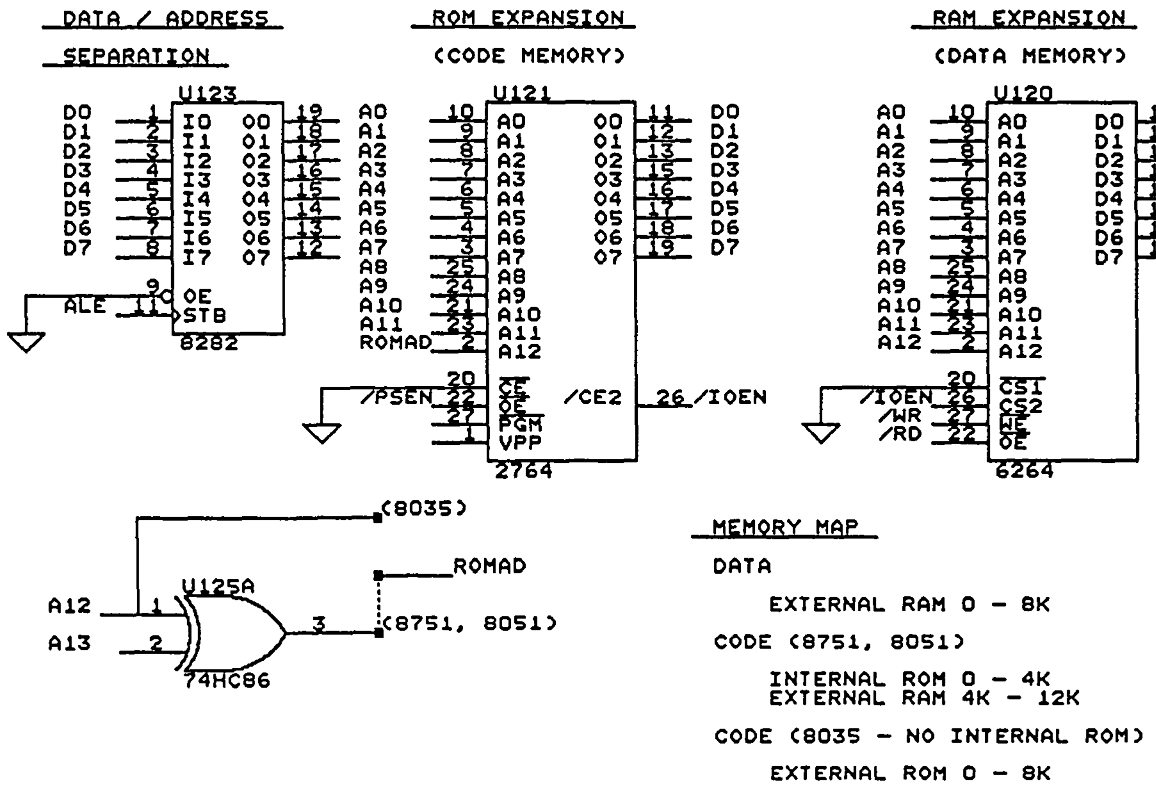


Figure 7.3 The optical module external memory circuit.

The first part of the circuit is the data / address separation logic. Here, the multiplexed data / address lines D0 through D7 inclusive are fed directly from the microcontroller (U122) to U123 (Intel 8282 straight octal latch) which acts as a demultiplexer separating off the address lines A0 through A7 inclusive. The time at which the multiplexed lines contain valid address information is indicated by the ALE (Address Latch Enable) signal directly from the microcontroller. In turn, the address lines are directed to both the external ROM (code only) memory and the external RAM (data only) memory along with the original (multiplexed) data lines D0 through D7 inclusive. The 8051 family microcontroller provides two separate strobes or chip enable signals to indicate in data time (as opposed to address time) that valid data information is available. The external ROM U121 (2764 +5 V only 8 kilobyte by 8 bit) is strobed by PSEN (Program Sense ENable) directly from the microcontroller. Analogously, the external RAM U120 (HM6264 static CMOS +5 V only 8 kilobyte by 8 bit) is strobed by RD (ReaD) during a read cycle and by WR (WRite) during a write cycle, both originating directly from the microcontroller. The timer output T1, instead of being used for its intrinsic function as a 16 bit timer, is used to provide a common external memory strobe signal IOEN (I/O ENable). Finally, the higher address lines A8 through A13 inclusive are used to fill in the remaining address control lines. In the case of both the RAM and the ROM, A8 through A11 inclusive directly drive four of the address lines. For the RAM, A13 is used to directly drive the remaining address line. For the ROM, a combination of A12 and A13 through U125A (74HC86 XOR) is used to generate the remaining address line ROMAD (ROM ADdress) according to the formula:

ROMAD = A12 XOR A13 for the 8751 and 8051 versions

ROMAD = A12 for the 8035 version. (7.1)

The communications interface is as good a place as any to continue the description of the digital circuitry of the optical module. The reader is referred to Figure 6.1 of the previous chapter, specifically to the modem chip U111. Three external signals are required by this chip to make it do anything worthwhile. These signals are labeled RXD, TXD, and MCLK. RXD and TXD are the "Receive (digital) Data" and "Transmit (digital) Data" signals respectively and tie directly into the 8751 microcontroller chip (U122 of Figure 7.1). The remaining signal, MCLK is the "Modem Clock" signal and is an exact 4.032 Mhz square wave clock generated by the timing chain logic shown in Figure 7.4 below:

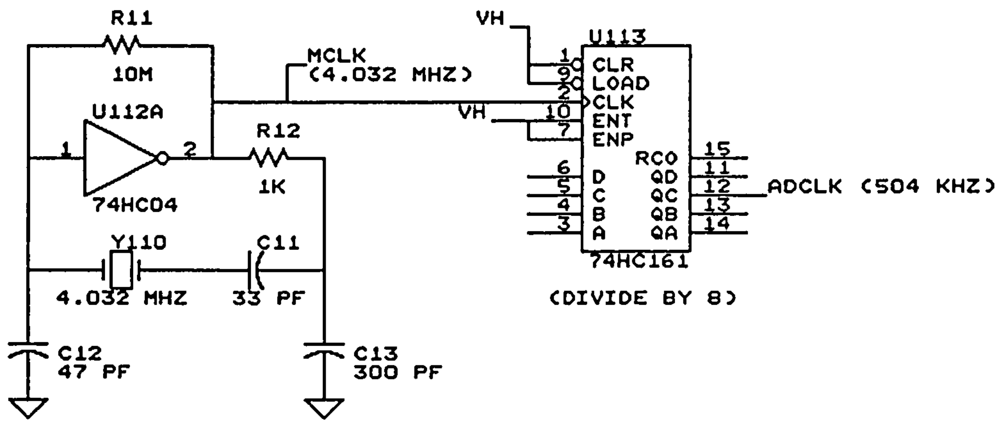
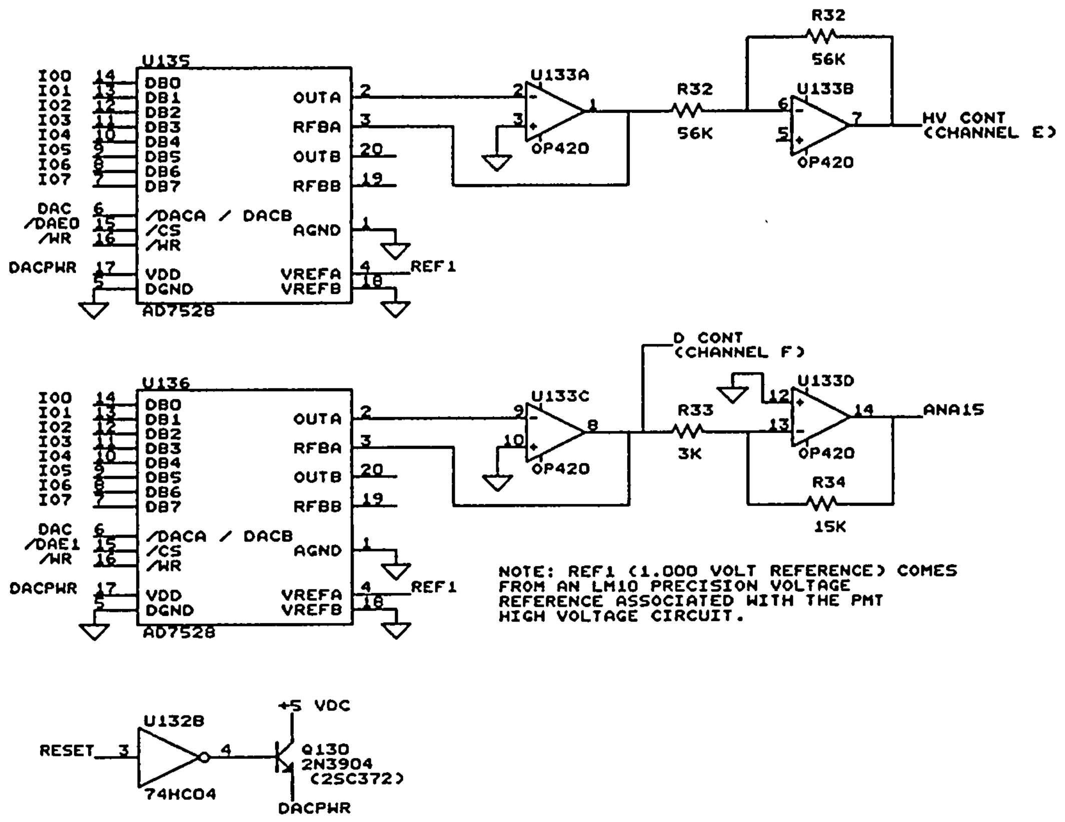


Figure 7.4 Optical module A to D and communications timing chain.

The 4.032 Mhz crystal frequency is needed by the modem chip in order to provide the right frequencies to meet the Bell 103 300 baud specification. A high speed CMOS gate, U112A (74HC04 inverter), with resistor R11 (10 MΩ) in parallel to bias the gate in its linear region, form the basis of a "high speed CMOS" self oscillating circuit that starts up upon feeling a power on transient. Resistor R12 (1 KΩ) provides an impedance that adds some additional phase shift in conjunction with capacitor C13 (330 pf). This has the effect of cutting out spurious high frequency oscillations isolating the gate output from the crystal network Y110 (4.032 Mhz) so that a clean square wave results. The value of R12 is chosen so that it will be roughly equal to the capacitive reactance of C13 at the frequency involved. Capacitors C12 (47 pf) and C13 form the load resistance for the crystal. Most crystals are cut for either 20 or 32 pf load capacitance. Using values larger than this, C12 and C13 swamp out the effects of temperature and supply voltage change on the input and output impedances. Since C13 is so much greater than this, capacitor C11 (33 pf) is placed in series with the crystal to act as the load for the crystal and thus assure an impedance match so that the crystal will not be loaded down. The result is the 4.032 Mhz square wave MCLK pulse train. This is fed both directly into the modem chip as well as into U113 (74HC161 asynchronous binary counter) where a divide by eight is performed generating the 504 Khz clock ADCLK which in turn, as a system clock, is tied directly into the ADC0816 (U130) 16 channel, 8 bit, analog to digital converter chip.

Figure 7.5 below documents the control circuitry interface. Two individual channels are controlled, the high voltage (PMT gain) and the (noise limiting) discriminator threshold. Both circuits use the Analog Devices AD7528 dual eight bit A to D converter. In each case, only one channel was used in this prototype to ensure that there was no crosstalk on such a critical matter as controlling the PMT voltage and establishing the noise cutoff level. This was considered especially critical since variation of either one of these parameters will affect the PMT's single's rate. As such, massive confusion could result if crosstalk occurred and one did not know which parameter had effectively changed the single's rate that one was monitoring.



NOTE: REF1 (1.000 VOLT REFERENCE) COMES FROM AN LM10 PRECISION VOLTAGE REFERENCE ASSOCIATED WITH THE PMT HIGH VOLTAGE CIRCUIT.

Figure 7.5 The (D to A) control interfaces.

U135 and U136 are the D to A converters for the high voltage and discriminator threshold channels (E and F) respectively. Both are not powered on immediately like the rest of the electronics in the optical module. The same RESET signal which is used to enable a controlled power up of the microcontroller is used to control the power applied to the D to A converters. RESET is applied to U132B (74HC02 inverter) whose inverted output is in turn used to switch on transistor Q130 (2N3904 or 2SC372). This switched power, labeled DACPWR, is then applied to the A to D converters. As soon as the microcontroller starts up, it goes into a loop initially "turning off" the high voltage supply so that no damage is accidently done to the PMT because some random high value was initially impressed upon the high voltage control channel upon system power up. Digital information is transferred directly from the microcontroller to the D to A converters by way of lines IO0 through IO7 inclusive. In both cases, the strobe signal DAC is used to select the channel in the D to A converter actually used. Since, when the microcontroller is actually writing to one of the D to A converters it can not be, at the exact same time, writing or reading memory, the address line A14 acts as the source for the control signal DAC. This is an example of the advantages of the microcontroller approach where one line can be effectively multiplexed in time to do two different things. To select out the time in which this I/O transfer will take place, the WR strobe directly from the microcontroller is used. The distinction as to which D to A device is to be used is made by the chip enable signals DAE0 and DAE1 mentioned before (Table 7.3). Finally, common to the two D to A converters is the signal REF1. This is provided outside the digital logic circuitry. It is a precision 1.000 V reference source for the D to A converters. It is produced by a National Semiconductor LM10 (precision voltage reference) associated with the PMT high voltage analog circuitry. As such, the range of voltages to be found from these two control outputs is from 0 to 1.000 V.

The high voltage control output, OUTA1, is fed into U133A (Precision Monolithic OP420 operational amplifier) where it is inverted and then fedback into the D to A converter as reference signal RFBA1. RFBA1 is in turn fed to a unity gain buffer U133B which directly drives the programmable high voltage supply of the PMT. Similarly, the discriminator threshold control output, OUTA2, is fed into U133C where it is inverted and then feedback into the D to A converter as reference signal RFBA2. RFBA2 is then used to directly control the level of the threshold discriminator. As was mentioned before, it was decided to directly monitor the actual levels that the high voltage and threshold discriminators actually settled into. In the case of the threshold discriminator, this feedback was established by amplifying the maximum 1.000 V level signal to a 0 to 5 V level range by in turn passing it through the 5 times amplifier U133D whose output ANA15 (ANAlog channel 15) is at the right level to be fed directly back into the A to D converter (U130). The feedback for the other channel comes from the PMT high voltage analog circuitry.

Figure 7.6 documents the sensor input interface. A single chip U130 (ADC0816 eight bit wide, 16 channel A to D converter) is used to convert all 15 assigned analog sensor channels, ANA0 through ANA14 inclusive, into digital input information which is directly conveyed to the microcontroller by way of lines IO0 through IO7 inclusive. The A to D converter starts the conversion process when it receives a START signal made up of the combination in U132 (74HC02 NOR)

$$\text{START} = \text{WR AND ADC} \tag{7.2}$$

where ADC is the (Analog to Digital Converter) strobe mentioned before (Table 7.3). U124 (Analog Devices AD584) forms a precision +5 VDC voltage reference. Timing pulses for the conversion process are supplied by the 504 Khz square wave clock ADCLK (Figure 7.4)[14].

[14] Which is just about nominal (500 Khz) according to the ADC0816 specification. See National Semiconductor Linear Handbook.

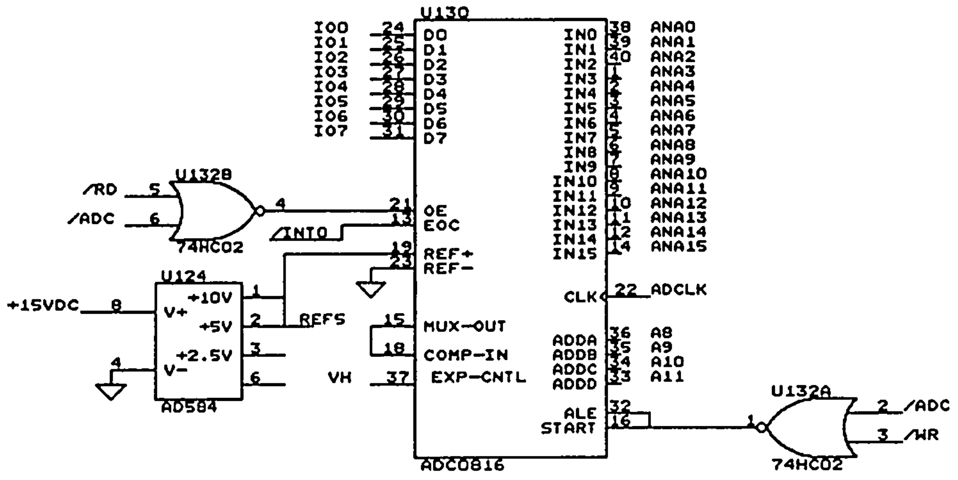


Figure 7.6 The (A to D) sensor interfaces.

When the conversion process is over, an end of conversion (EOC) signal is generated and applied directly to the microcontroller as INT0 (INTerrupt 0). Since a polling rather than interrupt philosophy was adopted, an actual interrupt is not generated in the microcontroller but rather, at this time, the microcontroller is dedicated to monitoring the sense of this input. When the microcontroller senses that the conversion is complete, then it starts the data read process upon receiving an OE (Output Enable) signal made up of the combination in U132B (74HC02 NOR)

$$\text{OE} = \text{RD AND ADC.} \tag{7.3}$$

The A to D converter includes a 16 channel multiplexer. The internal line COMP-IN (COMParison INput) is the input to the A to D section of the chip. An analog signal in the referenced range of 0 to +5 VDC will be converted to its eight bit digital equivalent. The internal line MUX-OUT (MUltipleXer OUTput) is the output of the 16 channel multiplexer section of the chip. A given channel is selected through the channel address inputs ADDA through ADDD which are tied directly into the microcontroller as address lines A8 through A11 inclusive strobed by the START signal. These two lines, COMP-IN and MUX-OUT are tied together to connect the two sections of the chip.

A calibration module[15] was deemed necessary so as to be able to assure what light levels were actually measured by the optical modules when deployed in the deep ocean to measure the actual attenuation length of light there and to do in situ calibration. A pulsed nitrogen laser with a diffuser / scintillator ball was used as the light source. An on-board photodiode was used to confirm light generation. Additionally, a mechanical light attenuator was used to provide, on command, a series of different known light levels. All of these requirements necessitated digital control circuitry quite similar to the optical module. As such the digital

[15] This is documented in detail in the pending (1989) dissertation of John Clem, Vanderbilt University.

logic circuitry of the optical module, after it was developed, was borrowed wholesale for the purposes of controlling the calibration modules. Figure 7.7 documents the adopted interface.

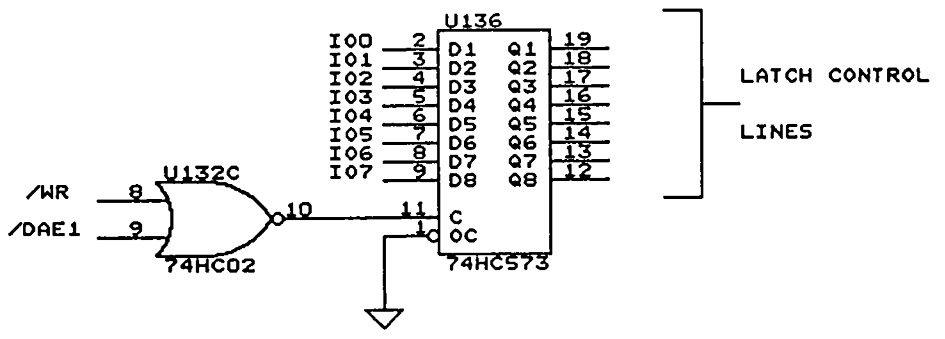


Figure 7.7 The calibration module interface.

There was no need for a threshold discriminator control so this (channel F) became the means by which the calibration module was controlled. Replacing the AD7528 D to A converter chip was a 74HC573 octal latch. The same time slice used before for this channel was provided by a 74HC02 NOR gate driving the EN (ENable) input of the latch

$$EN = WR \text{ AND } DAE1. \tag{7.4}$$

In this time, I/O lines IO0 through IO7 inclusive, provided up to eight control lines. Thus, writing a desired bit pattern (of up to $0FF_H$ or 256 combinations) to channel F provided control of the calibration module. What was particularly nice about this approach was that no changes at all were needed in software development and as such the calibration modules were treated on a completely equal basis with the optical modules. Order wire communications, etc. were all the same.

7.3 Diagramming the Flow of a Computer Program

Before documenting any of the software efforts it is important to establish some conventions. A computer program consists of a sequence of instructions occasionally interrupted by a branch or jump to some other location in memory where again a sequence is picked up. The flow chart is a tool for diagramming the flow of a computer program. It consists of only a few symbols connected together by flow arrows indicating the direction of flow out of one symbol and into another. The symbols are all boxes of sorts enclosing either a node label, a routine name, or some logical condition to be tested. The symbols are as follows:

Circle - this is used to indicate a node in the flow. For example, the beginning of a program or a routine typically will start with a node containing the name of the program or the routine. Later, if the flow chart is going to be too big to represent on a single piece of paper, a common node name may be given to the last point on the paper and the first point on the next paper to indicate a continuation of the flow. A node, at most, has only one input arrow and one output arrow associated with it.

Rectangle - this is used to indicate a code sequence. It may contain some indication of what the sequence is doing or it might contain the name of the routine called upon to perform the sequence. On the grossest scale, the computer program itself can be represented by one such symbol. A sequence has only one input arrow and one output arrow associated with it.

Diamond - this is used to indicate a decision that the program makes by testing a logic condition. The condition being tested is indicated inside the symbol. A single input arrow enters this symbol. As a decision, in which different directions in program flow are possible due to different results of testing the logic condition, more than one output arrow is associated with the symbol. It is usually best to limit the number of output arrows to two and to break down more complex decisions accordingly. The output arrows are labeled as to the results of the decision.

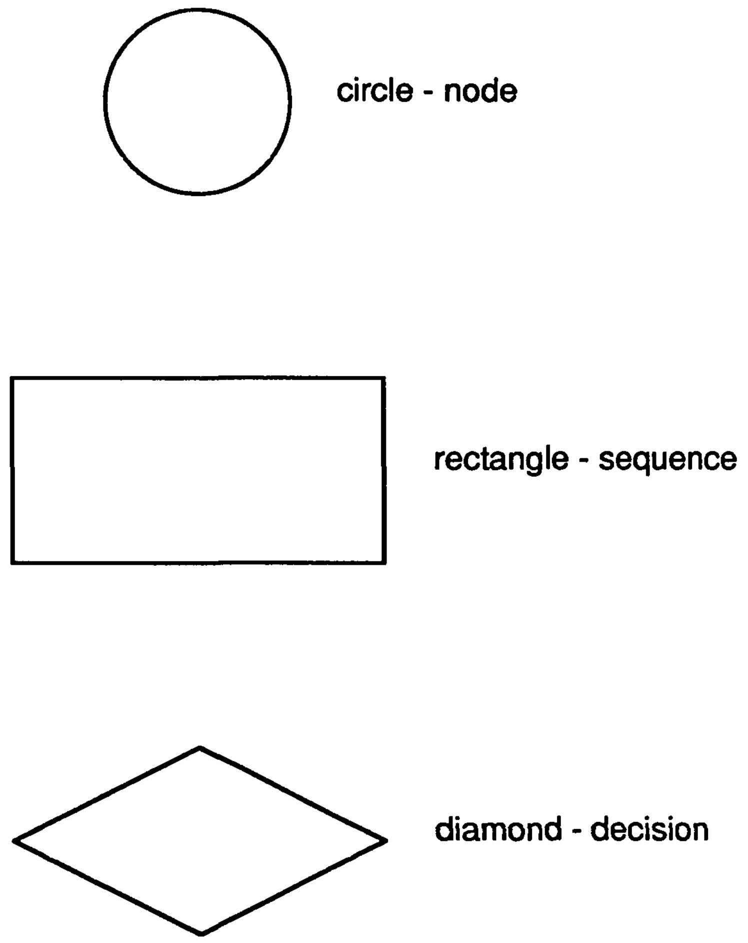


Figure 7.8 The three basic flow chart symbols

7.4 The Microcontroller Development System (MDS)

At the time initial development of an 8051 family microcontroller based system was undertaken in the DUMAND laboratory in Hawai'i, there were no real third party development systems available. INTEL corporation, the manufacturer of the 8051 microcontroller had two, a very crude ROM based system which required the use of a ROM in the development cycle and a microcomputer based system, known as the Microcontroller Development System (or MDS for short). Hewett Packard corporation also had a system but it cost about four times as much as the INTEL MDS.

The use of a ROM based development system, with no real way to create and store programs, would have been absurd to get any production work done. The need to use ROM's in the development cycle would have been almost as disastrous. In general, it is a good idea to set up a development system with the ability to do ROM simulation. In such a system, one does not have to go through all the effort of burning and erasing ROM's just to try a new change in code. It gets old fast and the ROM's themselves are soon used up by exceeding the practical limits of the number of times they can be recycled before they are worn out. ROM's are very important in many applications but they are a disaster in development because their use wastes so much development time. There are a number of ways to simulate ROM's. Generally speaking, these methods work on the variation of the theme of building a RAM circuit which can be written to under system initialization conditions by an external controller and then have the RAM locked out for reading under program execution. In this way, to the circuit under test, the RAM looks like ROM.

The alternative then, was to obtain an INTEL Microcontroller Development System. This system, consists to two parts. The first part is the MDS-225A smart terminal. It is an 8085 microprocessor based bench top computer with a somewhat unique (but ASCII) keyboard and a single sided 8" floppy-disk. The operating system running on it is proprietary to INTEL and is called ISIS-II. ISIS-II is akin to a number of eight bit operating systems somewhat like CP/M, but because it is unique to INTEL, this means that the development software running on it is restricted to INTEL. The second part is the 8051 in-circuit emulater (ICE-51). This device consists of a series of boards which plug into the MDS-225A. A ribbon cable then leads out to a special 40 pin plug which plugs directly into the circuit under test as if it were the microcontroller. The ICE-51 provides both ROM simulation and 8051 family microcontroller simulation. ROM simulation is accomplished by loading an assembled program into the memory of the ICE-51. Microcontroller simulation is accomplished by letting the control program of the ICE-51 execute the program loaded into the memory. A crude ability to set two breakpoints is provided and a symbol table is accessible. It is interesting to note that while INTEL went out of its way to create a microcontroller with totally separate DATA and CODE memories, it combines the symbol tables from both limiting the usefulness of having a symbol table in tracing program execution because of the high probability of two different symbols appearing with the same value, the most recent one clobbering the former one.

Once the decision was made to go with the 8751 microcontroller family, the effort began to purchase an MDS-225A and ICE-51 with the necessary support software. Eventually, this was accomplished, but only after a nine month exercise of frustration with University of Hawaii purchasing procedures. Thus, development got started later than originally desired.

The system came with a reasonable screen oriented editor (AEDIT), an 8051 cross assembler (MCS51), a linker (RL51), a series of communications programs (ACL commands ONLINE, SEND, UPLOAD, AND DNLOAD) linking the MDS to the VAX computer at HDC, and the driver for the in-circuit emulater (ICE51). Thus, the development cycle would go something like Figure 7.9

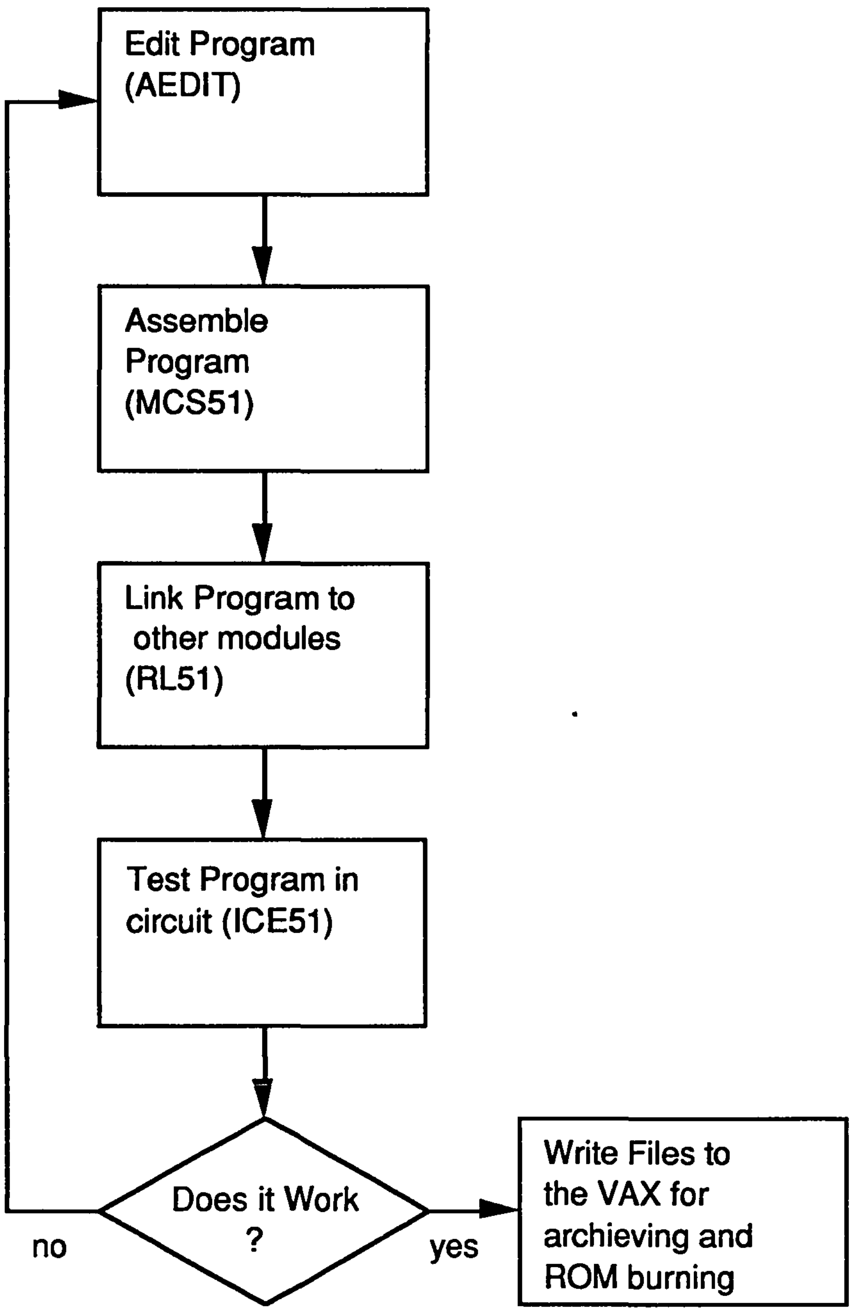

Edit Program (AEDIT)
Assemble Program (MCS51)
Link Program to other modules (RL51)
Test Program in circuit (ICE51)
Does it Work ?
no
yes
Write Files to the VAX for archieving and ROM burning

Figure 7.9 MDS development cycle.

The MCS-51 (cross) assembly language[16] is a rather unusual mixture. On the one hand, in terms of the assembly options that one can choose from (and, in a certain sense, must at least partially understand in order to minimally satisfy the assembler to do something), it is a very rich package. There is much to choose from. This includes a macro facility, known as MPL (Macro Processor Language), with strong facilities for defining functions as well as a good collection of control functions (such as IF-THEN-ELSE, REPEAT, and WHILE). However, the macro definition functions follow a syntax that differs from the more popular syntax found in common in MACRO-11 (the PDP-11 and VAX assemblers) and the CP/M standard MACRO-80.

On the other hand, the instruction set of the 8051 family microcontrollers is the worst I have ever seen. It is unnecessarily burdened with "single byte" instructions which do almost nothing and use up the instruction set space so that there is no room for even some of the more common instructions one would expect on a microcomputer else where (such as the Motorola 6800, 6809, and 68000 series or the Intel 8080, 8085, and 8086 series). Altogether, MCS-51 is a poor development system mainly because of the limited instruction set. In order to be able to program this microcontroller to the relatively sophisticated level that was demanded by the optical modules, calibration modules, power controller, and the Ersatz SBC, a more malleable programming system was needed. This I accomplished by developing a language based upon the MPL macro facility called UHPS (Underwater Hawai'i Programming Language). UHPS is the subject of Chapter 9 and its details are left for there. Suffice it to say that UHPS is a structured language akin to C and PASCAL and that it has many macros added that allow it to do things that ought to be intrinsic to the instruction set and yet are not.

One of the great advantages of structured languages is that they readily lend themselves to the development of independent modules that

[16] See MCS-51 Macro Assembler User's Guide

can be easily tested in isolation from the application that they are intended for. One effective class of such modules in the hardware intensive environment of the SPS are the device drivers. These are the routines which directly interface with and control the hardware such as an AtoD controller, etc. With the aid of simple laboratory test instruments such as bench power supplies, an oscilloscope, and a digital voltmeter one is able to set up an independent testing and debugging environment for the device driver. A small throw away routine is written which is used for passing information either to or form the device driver under test as if it were embedded in the applications program. The disadvantage of this approach is the need for throw away code. However, these are usually very straight forward and typically only variations upon a common theme and as such, once one or two are working, it is not a really involved job to bring up others. The main advantage of this approach is that the modules are independent of one another and they have been thoroughly debugged before being linked into the applications program. In short, one tames the beast by dividing and conquering.

7.5 Optical Module Program Description

The string optical module program (SOM) is used to provide control and communications over the common party line string for both the optical and calibration modules. With seven optical modules, two calibration modules, and the ability to communicate with only one module at a time, most of the time this program spends "listening" to the string for commands. The principle objectives of this program are (1) to provide party line communications, (2) to provide control of the PMT high voltage and threshold discriminator level, and (3) to report back values of various analog channels as well as the PMT single's rate.

The program SOM is made up of some 23 separate routines, not counting the modules associated directly with the UHPS language such as the telecommunications and timing routines. The routines found in SOM logically divide themselves into eight classes as is summarized in Table 7.4 below:

Table 7.4

Summary of SOM Routines

Class	Name	Description
Main:		
	SOM	Main program
	INITIAL	Initialize stack and 300 baud serial port
Communications control:		
	CREPLY	Echo back valid command
	RREPLY	Send back single's rate value
	SYNCH	Synchronize on first byte of command string
Parser:		
	CMDPOLL	Parse the command string for a valid command
Table handler:		
	ANAPOLL	Poll all (16) analog channels
	ANLKUP	Look up (read) analog value from table for specified channel
	UPDATE	Update (write) analog value to table for specified channel
Command executive:		
	CMDSERV	Service the command request
	RRATE	Read the PMT single's rate value

Table 7.4 (continued)

Summary of SOM Routines

Class	Name	Description
Device drivers:		
	ATOD	Drives AtoD converter for specified channel
	DTOA	Drives DtoA converter for specified channel
	MODID	Look up this module's identification number
	SETDSCR	Set the threshold of the discriminator
	SETHV	Set the PMT's high voltage
Error handling:		
	CMDERR	Report a command error
	DEVERR	Report a device error
	FORERR	Report a format error
Glue:		
	ISALNO	Check if byte is alphanumeric
	ISHEX	Check if byte is a hexidecimal number
	RAMPDN	Ramp down the PMT's high voltage
	RAMPUP	Ramp up the PMT's high voltage

A complete description of this program would take up an inordinate amount of space. For the interested reader, Appendix 4 is a complete listing of this program including many in line comments. The program is written in the language UHPS which is detailed in Chapter 9. This chapter must be studied in order for the reader to be able to really follow the program listing in the detail necessary to make intelligent changes. What follows is a greatly simplified description of some of the salient features of the program.

Understand that much of the important implementation detail is left out of this description so as to make the overall thrust of the program comprehensible.

The "main" or "root" level of SOM is flowcharted in Figure 7.10 below. Upon powering up, the microcontroller first executes an initialization routine INITIAL. This routine, which is common to both the optical module and power module programs, is flowcharted in Figure 7.11. It is initiates the system stack and sets up the bidirectional 300 baud serial communications port.

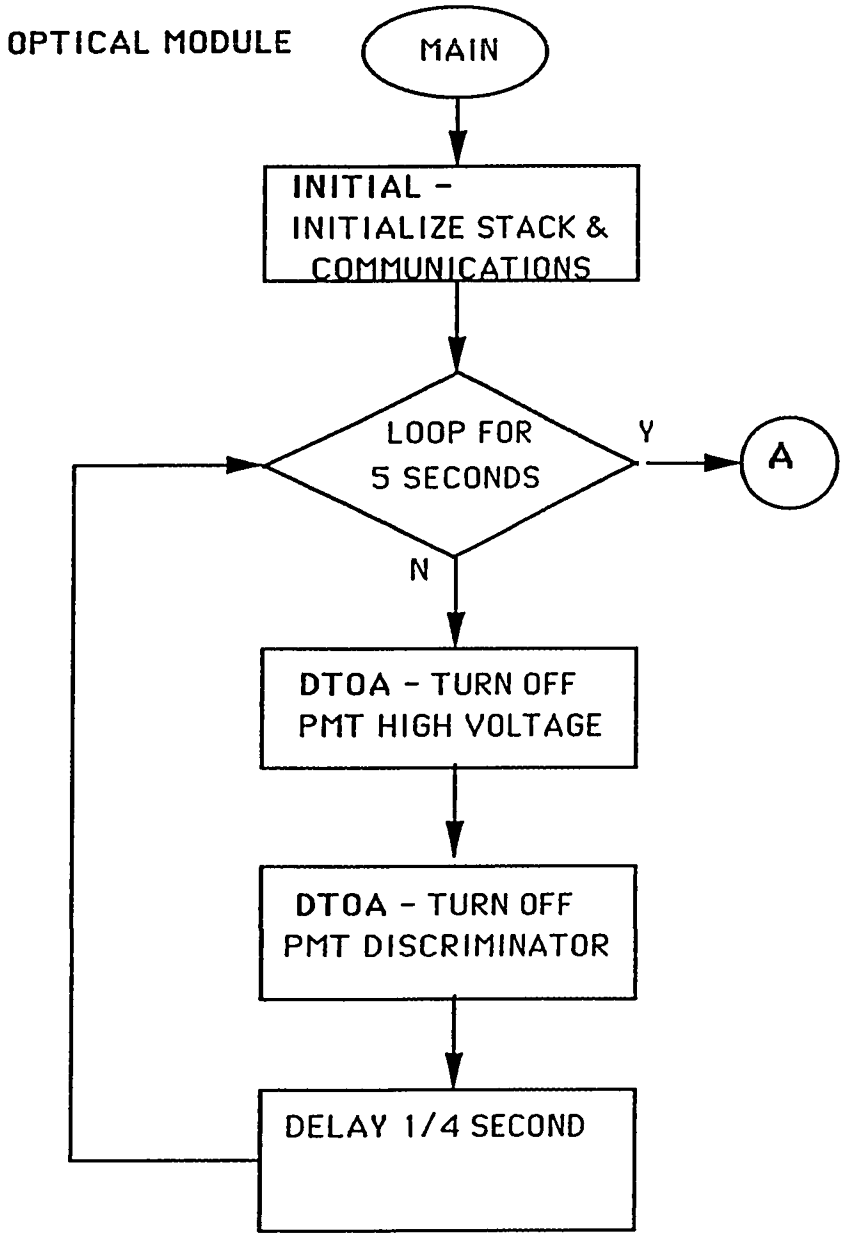


Figure 7.10 String Optical Module main program.

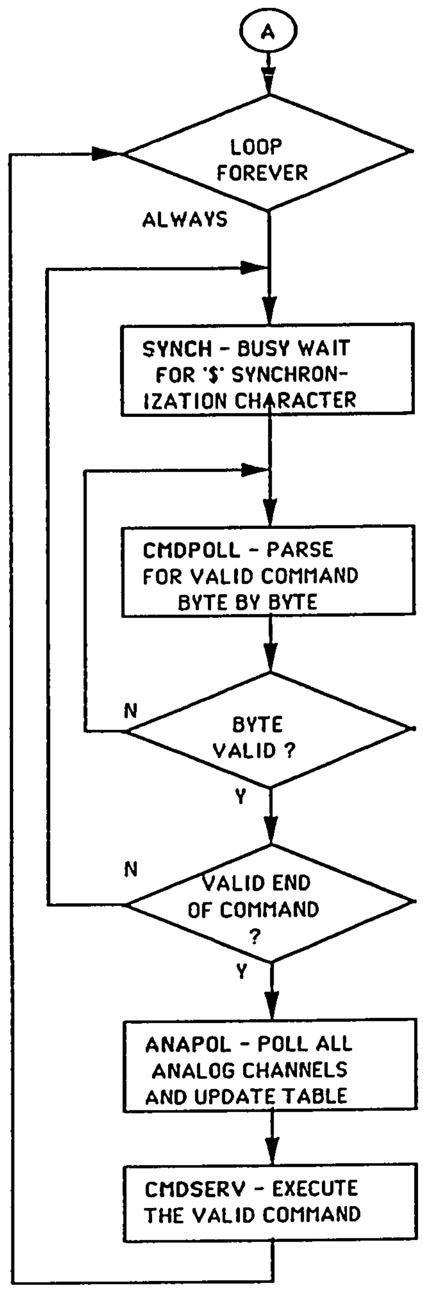

A
LOOP FOREVER
ALWAYS
SYNCH - BUSY WAIT FOR '$' SYNCHRONIZATION CHARACTER
CMDPOLL - PARSE FOR VALID COMMAND BYTE BY BYTE
N
BYTE VALID ?
Y
N
VALID END OF COMMAND ?
Y
ANAPOL - POLL ALL ANALOG CHANNELS AND UPDATE TABLE
CMDSERV - EXECUTE THE VALID COMMAND

Figure 7.10 (Continued) String Optical Module main program.

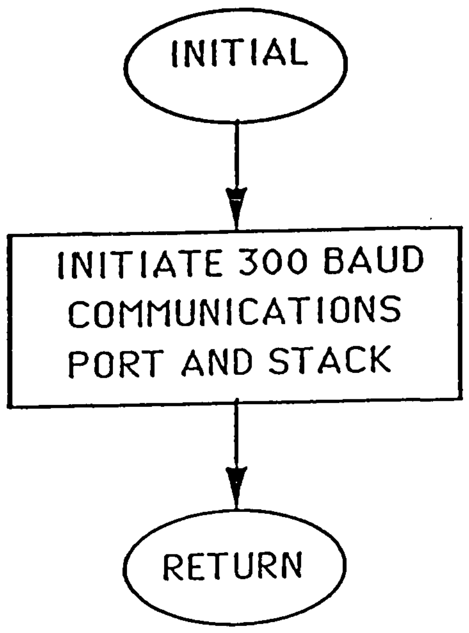


Figure 7.11 Common microcontroller initialization routine.

After completing the initialization, the program goes into loop where by it sends values to both of the DtoA channels repeatedly turning off the PMT high voltage supply and maximizing the threshold level of the discriminator. This is done to place everything is a known, safe condition so that no damage is done to the PMT. The loop is repeated every 1/4 of a second for a total of five seconds to be sure that everything has had power long enough so as to become stable and is under control. An example of a device driver routine, DTOA, is flowcharted in Figure 7.12.

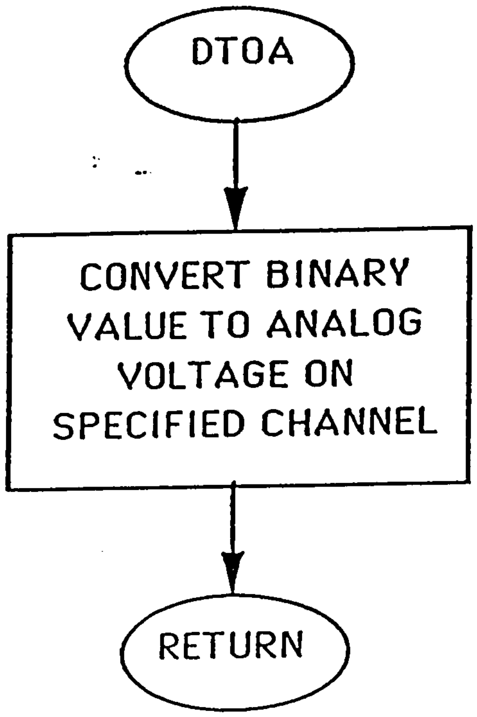


Figure 7.12 Flowchart of the digital to analog device driver DTOA.

At this point, the program enters into an infinite loop. The function of this loop is to satisfy the main objectives of the program. The first objective, the handling of party line communications, is achieved in part by the routine SYNCH. This routine is common to both the optical module program and the power module program. It is flowcharted in Figure 7.14. All commands sent down to the optical and calibration modules are of the form:

$<DEV><DEV><CMD0><CMD1>{<DAT0><DAT1>}<CR> where

<DEV>	= device indentification number (1..7 for optical modules and A..B for calibration modules)
<CMD0>	= command byte 0 - either 'R' for read or 'W' for write
<CMD1>	= hexidecimal channel number (0..F) or 'R' for single's rate
<DAT0>	= optional first data byte
<DAT1>	= optional second data byte

Figure 7.13 Optical module command line

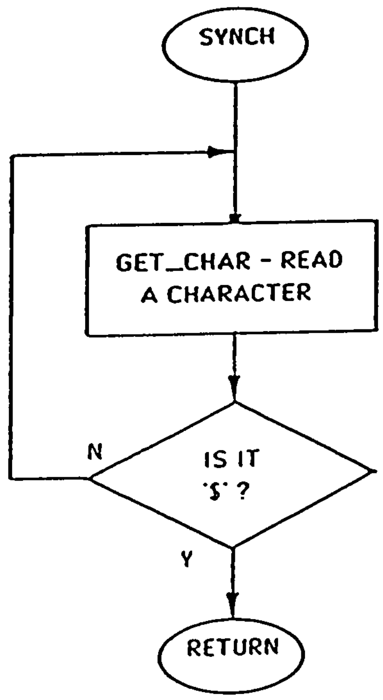


Figure 7.14 Command line synchronization routine SYNCH

SYNCH is a busy wait loop which continues to look at successive characters in the serial communications stream for the special synchronization character '$'. When this character is found, the SYNCH is exited and program control is passed on to the main parser routine CMDPOLL. This is the most complicated routine in the entire program. A highly simplified flowchart is given in Figure 7.15.

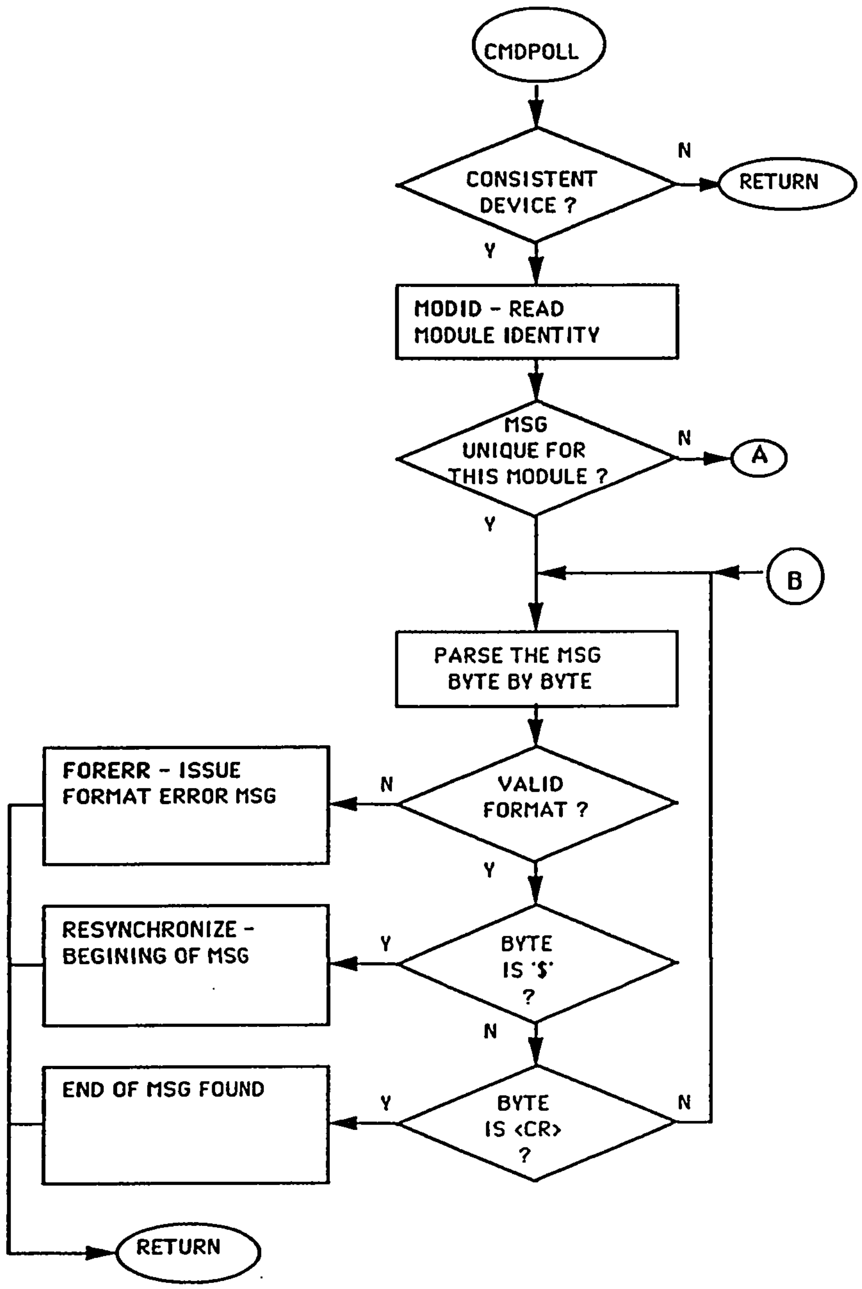

CMDPOLL
CONSISTENT DEVICE ?
N
RETURN
Y
MODID - READ MODULE IDENTITY
MSG UNIQUE FOR THIS MODULE ?
N
A
Y
B
PARSE THE MSG BYTE BY BYTE
FORERR - ISSUE FORMAT ERROR MSG
N
VALID FORMAT ?
Y
RESYNCHRONIZE - BEGINING OF MSG
Y
BYTE IS '$' ?
N
END OF MSG FOUND
Y
BYTE IS <CR> ?
N
RETURN

Figure 7.15 The main command line parsing routine CMDPOLL

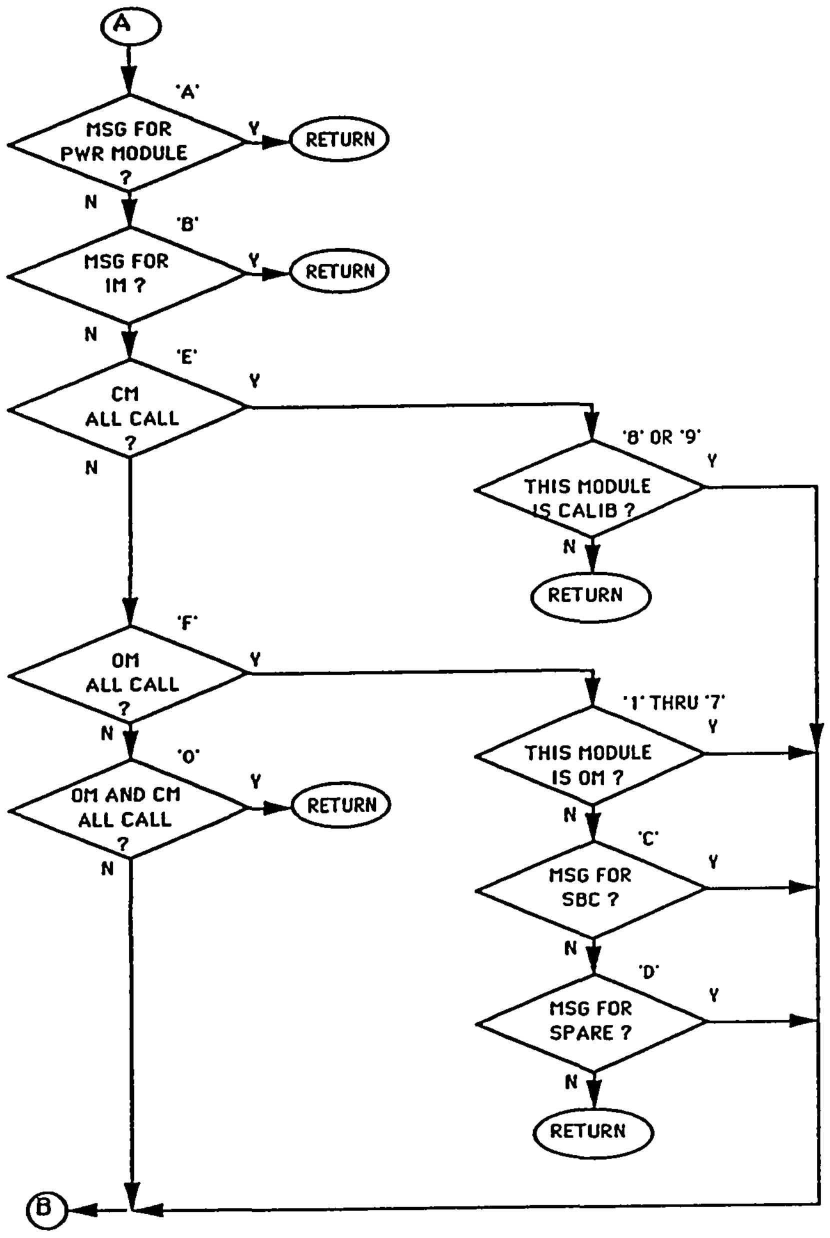


Figure 7.15 (Continued) The main command line parsing routine CMDPOLL

Basically, CMDPOLL looks at each byte in turn after a synchronization byte is found. It parses the list of bytes to be sure that they adhere to the defined command line format above. If, at anytime, a synchronization byte is encountered, the appropriate flag is set and the routine is exited at which point SOM loops back and calls it again. If, in the course of parsing, a syntax error is encountered, the routine is exited at which point SOM loops back again to the synchronization routine SYNCH searching for the beginning of the next command string. Finally, should a proper command for the particular module executing SOM be encountered, CMPOLL is exited without any error conditions and SOM then calls ANAPOLL.

ANAPOL, which is flowcharted in Figure 7.16, is the main ANALOG table handing routine. When entered, the first of 16 channels is pointed to and the ATOD routine called returning the analog value. In turn, UPDATE then places this value into ANALOG. The channel number is incremented and the process repeated until all of the channels have been updated. This was done in part because it was easier to program. The time it takes to loop through and update all 16 channels is fast in comparison to the time it takes to perform 300 baud communications.

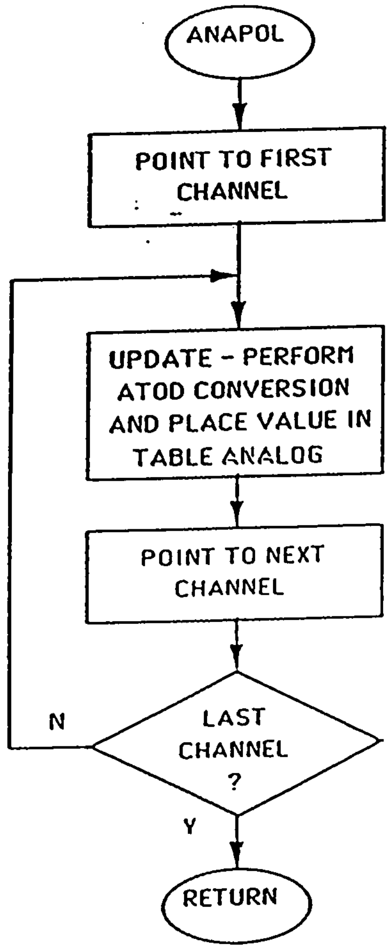


Figure 7.16 The main ANALOG table handler ANAPOL

Finally, Figure 7.17 flowcharts CMDSERV. This routine takes the command parsed out by CMDPOLL and checks to see if it is a lexically correct command. If it is not, a error reply message is generated by CMDERR. If it is correct, then the appropriate device driver is called either returning a requested analog value or the single's rate or changing the setting of the either the PMT high voltage (SETHV) or the threshold level of the discriminator (SETDSCR). In the later two cases, the new settings are automatically read on the appropriate analog channel and reported back as part of the reply message so that the operator can see what the real settings are and not just assume that the requested settings are correct.

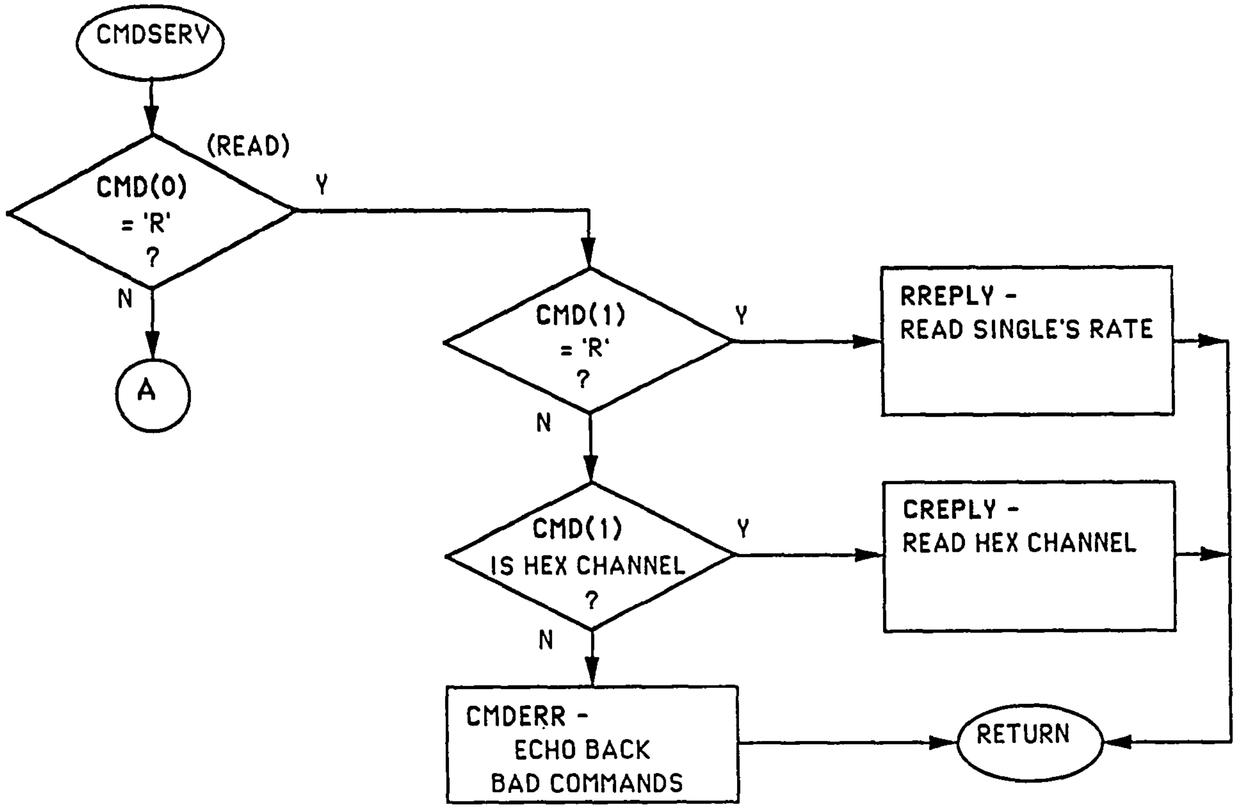

CMDSERV
(READ)
CMD(0) = 'R' ?
Y
N
A
CMD(1) = 'R' ?
Y
N
RREPLY - READ SINGLE'S RATE
CMD(1) IS HEX CHANNEL ?
Y
N
CREPLY - READ HEX CHANNEL
CMDERR - ECHO BACK BAD COMMANDS
RETURN

Figure 7.17 The command executive CMDSERV

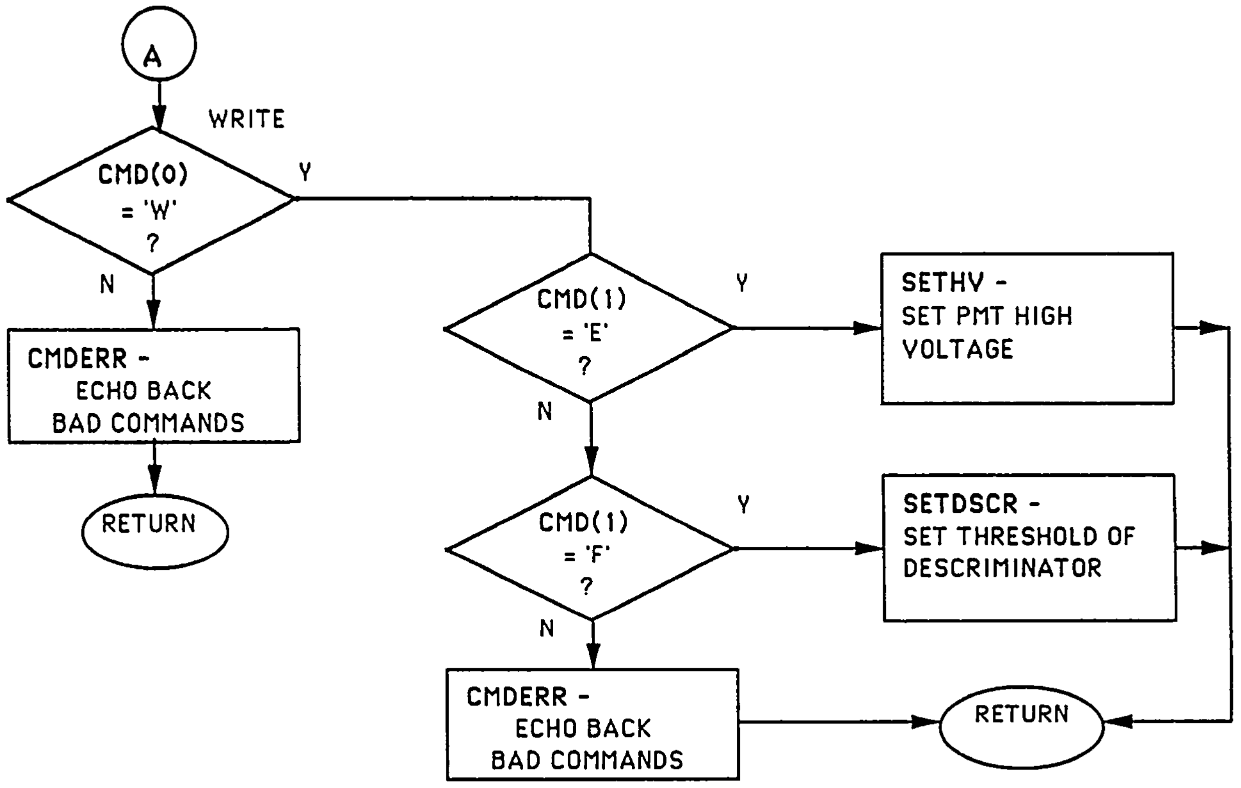

A
WRITE
CMD(0) = 'W' ?
Y
N
CMDERR - ECHO BACK BAD COMMANDS
RETURN
CMD(1) = 'E' ?
Y
N
SETHV - SET PMT HIGH VOLTAGE
CMD(1) = 'F' ?
Y
N
SETDSCR - SET THRESHOLD OF DESCRIMINATOR
CMDERR - ECHO BACK BAD COMMANDS
RETURN

Figure 7.17 (Continued) The command executive CMDSERV

CHAPTER 8 – Instrument Building III – The Design of the SBC Power Distribution System Microcontroller Circuit

8.1 Introduction

This Chapter is the third of a series of five on instrument building of the Short Prototype String. The main concern of this chapter is to document the hardware and software of the microcontroller based power distribution circuit. The microcontroller used in this circuit is identical to that used in the optical module control circuitry, namely the INTEL 8051 family. Chapter 7 which documents the optical module control circuitry discusses in some detail this microcontroller, its selection, peculiarities, and development environment and as such need not be duplicated here. The language used to program the microcontroller is again the same language as used in the optical module, namely UHPS, which is the subject of Chapter 9. The hardware is documented here in complete detail and the software is outlined with the complete listing, including in line comments, making up Appendix 5.

8.2 Power Distribution Circuit Description

The need existed to dynamically control the distribution of +48 VDC feeder power on an individual basis to the different instruments on the SPS. As such, as part of the SBC, an intelligent circuit, also based upon the INTEL 8051 family microcontroller was developed. This circuit, whose 300 baud communications was logically wired in parallel to the string, monitors exactly the same commands from the ship that the string does. The power distribution control circuitry consisted of four basic circuits: (1) 8751 microcontroller, (2) external memory, (3) communications interface, (4) analog to digital control and timing chain, and (5) parallel I/O power latch control.

Again, we start the description of the digital circuitry with its central controller, namely the INTEL 8751 microcontroller. This is documented in Figure 8.1 below:

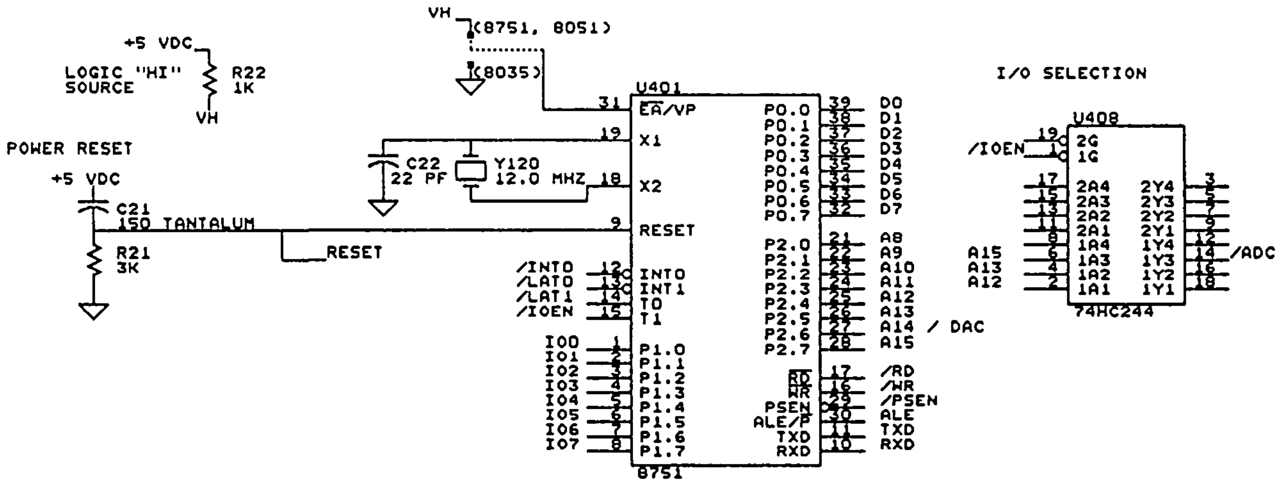


Figure 8.1 The power module INTEL 8751 based microcontroller circuit.

The 8751 microcontroller (U401) is a 40 pin chip requiring a single +5 VDC supply. Timing is generated by directly connecting a 12.0 Mhz crystal (Y120) across the two pins XTAL1 and XTAL2 along with a decoupling capacitor C22 (22 pf). Internally, the main control cycle timing of 1 Mhz is derived through a divide by 12 circuit. This same clock is used for the two 16 bit timers (T0 and T1), one of which is ultimately used to generate the baud rates for the two serial I/O ports (RXD and TXD). Two levels of interrupts are available ($\underline{\text{INT0}}$ and $\underline{\text{INT1}}$) as well as the write ($\underline{\text{WR}}$) and read ($\underline{\text{RD}}$) memory control pulses. These last eight lines constitute parallel I/O port 3 of the 8751.

System start up is accomplished with the resistor and capacitor combination made up of C21 (a 150 pf tantalum) and R21 (3 kΩ) tied directly into the RESET line. Upon power up, C21 feels the +5 VDC supply and is charged thus pulling the RESET line high. In time, R21 bleeds off C21 so that the RESET line goes low. The power on reset requirements for the 8051 microcontroller are such that the RESET line should be held high for at least one ms after the power supply stabilizes to allow the oscillator to stabilize. The microcontroller remains dormant until the line goes low. The other lines of the microcontroller assume their initialization states within 24 oscillator periods. Upon the RESET line going low, a sequence is initiated which takes about 12 oscillator periods before ALE is generated (permitting dialog with the memory) and normal operation begins with the program counter pointing to the first instruction at absolute program memory location OOOOH.[1] The $\underline{\text{EA}}$ (External Address NOT) line is used to tell the microcontroller whether it is an 8751 or 8051 version (logic high condition) of the microcontroller which has some on-board ROM with a program in it or not as is the case for the 8035 version (logic low condition). This way, when the microcontroller starts up, it looks in the right place to find its controlling program. The symbol VH stands for a common logical "high" line that can be used in all cases where a digital (+5 VDC) high is needed. It is formed by tying a 1 kΩ resistor into the +5 VDC supply. In this way, less

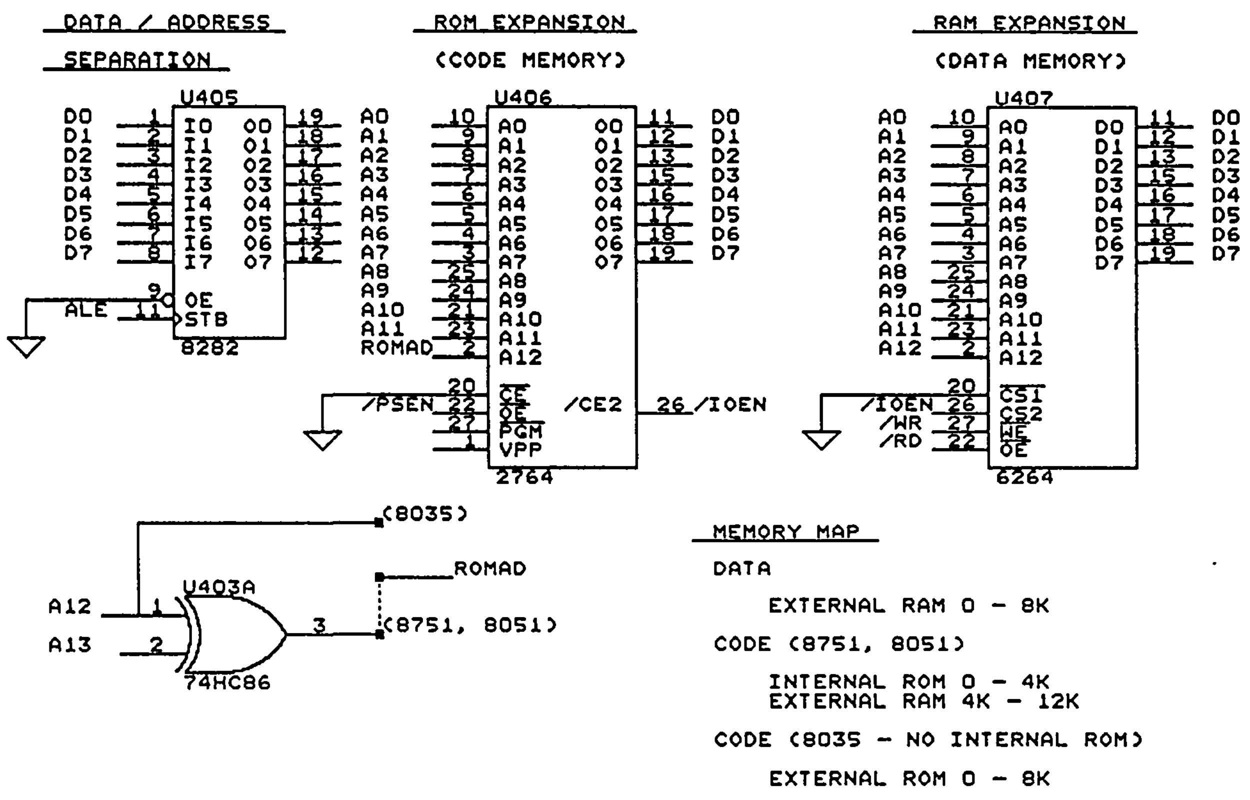
[1] MCS-51 User's Manual, p.2-28.

current is drawn by the chips than if they were directly tied in. This also means that should an over voltage occur, these inputs which are more sensitive than the power inputs, are afforded a degree of protection through current limiting.

The eight parallel I/O lines comprising port 0 are dedicated as the data transfer lines D0 through D7 inclusive. Actually, these lines are multiplexed in time so that they also act as the lower eight address selection lines for external memory. The ALE (Address Latch Enable) output of the microcontroller is the strobe signal for differentiating these two roles. The eight parallel I/O lines comprising port 1 are dedicated as the principle I/O lines for transferring information from the microcontroller to the different D to A channels and from the different A to D channels to the microcontroller. These are labeled IO0 through IO7 inclusive.

The eight parallel I/O lines of port 2 basically are divided into two different sections. The lines labeled A8 through A10 are used to address the ADC0808 (U402) 8 channel A to D converter picking out one of 8 possible channels. Lines A12, A13, and A15 are used (among other things) as inputs to one half of U131 (74HC244 non-inverting bi-quad tri-state buffer) to generate a series of chip enable lines, only one of which is used, $\underline{\text{ADC}}$ (A to D Control). These are separated in time from other uses of some of these lines by the strobing with the timer output T1 (labeled $\underline{\text{IOEN}}$ for I/O Enable).

As mentioned before in the case of the optical module circuitry, the 8051 actually has two 64 kilobyte memory spaces which have absolutely nothing to do with each other. The two separate memory spaces are called the "code memory" and the "data memory". The implementation of an external memory becomes essential because the internal limit of 80 data bytes is easily exhausted just to handle program variables. Figure 8.2 documents the external memory circuit:



MEMORY MAP

DATA

EXTERNAL RAM 0 – 8K

CODE (8751, 8051)

INTERNAL ROM 0 – 4K
EXTERNAL RAM 4K – 12K

CODE (8035 – NO INTERNAL ROM)

EXTERNAL ROM 0 – 8K

Figure 8.2 The power module external memory circuit.

The first part of the circuit is the data / address separation logic. Here, the multiplexed data / address lines D0 through D7 inclusive are fed directly from the microcontroller (U401) to U405 (INTEL 8282 straight octal latch) which acts as a demultiplexer separating off the address lines A0 through A7 inclusive. The time at which the multiplexed lines contain valid address information is indicated by the ALE (Address Latch Enable) signal directly from the microcontroller. In turn, the address lines are directed to both the external ROM (code only) memory and the external RAM (data only) memory along with the original (multiplexed) data lines D0 through D7 inclusive. The 8051 family microcontroller provides two separate strobes or chip enable signals to indicate in data time (as opposed to address time) that valid data information is available. The external ROM U406 (2764 +5 V only 8 kilobyte by 8 bit) is strobed by $\underline{\text{PSEN}}$ (Program Sense ENable) directly from the microcontroller. Analogously, the external RAM U407 (HM6264 static CMOS +5 V only 8 kilobyte by 8 bit) is strobed by $\underline{\text{RD}}$ (ReaD) during a read cycle and by $\underline{\text{WR}}$ (WRite) during a write cycle, both originating directly from the microcontroller. The timer output T1, instead of being used for its intrinsic function as a 16 bit timer, is used to provide a common external memory strobe signal $\underline{\text{IOEN}}$ (I/O ENable). Finally, the higher address lines A8 through A13 inclusive are used to fill in the remaining address control lines. In the case of the both the RAM and the ROM, A8 through A11 inclusive directly drive the four of the address lines. For the RAM, A12 is used to directly drive the remaining address line. For the ROM, a combination of A12 and A13 are used to generate the remaining address line ROMAD (ROM ADdress) according to the formula:

$$\text{ROMAD} = \text{A12 XOR A13} \quad \text{for the 8751 and 8051 versions} \tag{8.1a}$$

$$\text{ROMAD} = \text{A12 for the 8035 version.} \tag{8.1b}$$

The next circuit to consider is the communications interface. This is a fairly simple affair in which the TTL level microcontroller serial communications lines RXD and TXD, "Receive (digital) Data" and "Transmit (digital) Data" signals respectively, are translated into RS-232C level lines through the use of level translators U411A (1489 EIA receiver) and U412A (1488 EIA Driver). The communications lines RXD and TXD tie directly into the 8751 microcontroller chip (U401 of Figure 8.1). This is documented below in Figure 8.3.

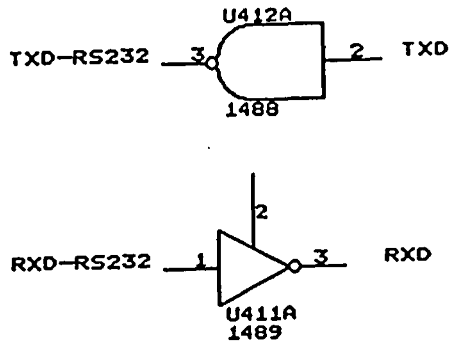


Figure 8.3 Serial communications in the power module.

The idea is that within the String Bottom Controller, the different serial communications paths, namely from the fiber-optics cable, the string, and the power module all to the central brain of the String Bottom Controller[2] would conform to a normal RS-232C specification so that in system

2 The subject of Chapter 7.

development and testing, any one of the communications pathways could be broken and a terminal attached to either end to monitor and isolate communications problems.

An eight channel, eight bit wide, analog to digital circuit was provided initially for sensor monitoring in the Erzatz String Bottom Controller cruise. This was later set aside for a larger system run directly by the SBC microprocessor system. This system is based upon the National Semiconductor ADC0808 A to D chip which is virtually identical to the ADC0816 used in the optical module circuit only it has eight instead of 16 channels. Figure 8.4 below documents this circuit.

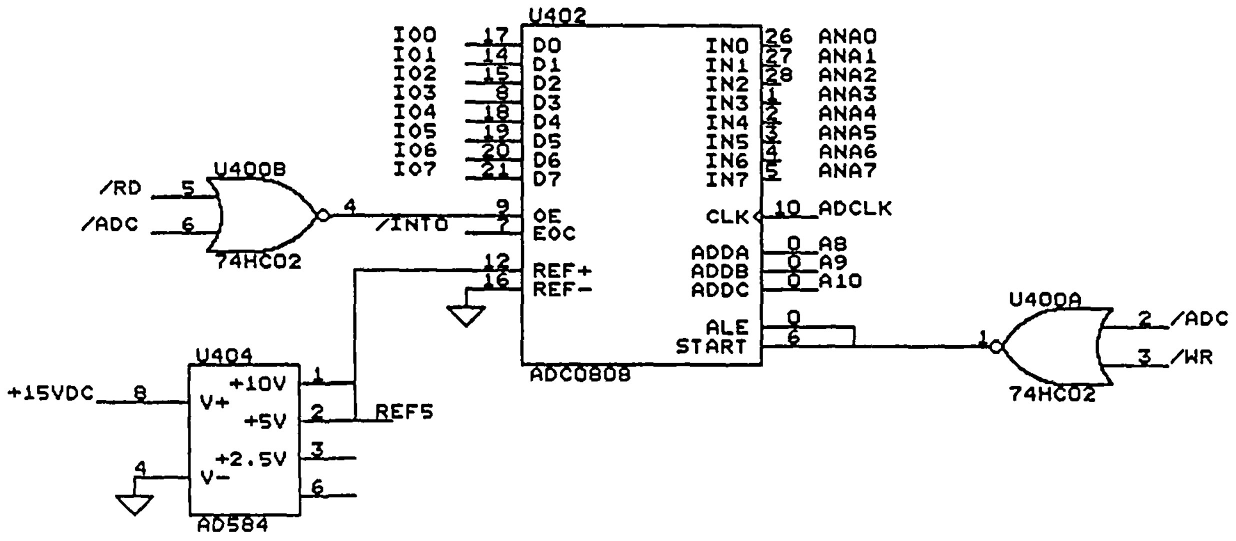


Figure 8.4 Power module A to D circuit.

The ADC0808 A to D converter needs a 500 KHz clock. This is provided by the high speed CMOS gate, U413A (74HC04 inverter), with resistor R11 (10 MΩ) in parallel to bias the gate in its linear region, form the basis of a "high speed CMOS" self oscillating circuit that starts up upon feeling a power on transient. Resistor R12 (1 KΩ) provides an impedance that adds some additional phase shift in conjunction with capacitor C13 (330 pf). This has the effect of cutting out spurious high frequency oscillations isolating the gate output from the crystal network Y110 (4.032 MHz) so that a clean square wave results. The value of R12 is chosen so that it will be roughly equal to the capacitive reactance of C13 at the frequency involved. Capacitors C12 (47 pf) and C13 form the load resistance for the crystal. Most crystals are cut for either 20 or 32 pf load capacitance. Using values larger than this, C12 and C13 swamp out the effects of temperature and supply voltage change on the input and output impedances. Since C13 is so much greater than this, capacitor C11 (33 pf) is placed in series with the crystal to act as the load for the crystal and thus assure an impedance match so that the crystal will not be loaded down. The result is the 4.032 MHz square wave MCLK pulse train. This is feed both directly into U414 (74HC161 asynchronous binary counter) where a divide by eight is performed generating the 504 KHz clock ADCLK which in turn, as a system clock, is tied directly into the ADC0808 (U402) 8 channel, 8 bit, analog to digital converter chip.

This single chip U402 is used to convert all 8 analog sensor channels, ANA0 through ANA7 inclusive, into digital input information which is directly conveyed to the microcontroller by way of lines IO0 through IO7 inclusive. The A to D converter starts the conversion process when it receives a START signal made up of the combination in U400A (74HC02 NOR)

$$\text{START} = \text{WR AND ADC} \tag{8.2}$$

where ADC is the (Analog to Digital Converter) strobe mentioned before. U404 (Analog Devices AD584) forms a precision +5 VDC voltage reference.

Timing pulses for the conversion process are supplied by the 504 Khz square wave clock ADCLK (Figure 8.5). Figure 8.4 documents the sensor input interface.

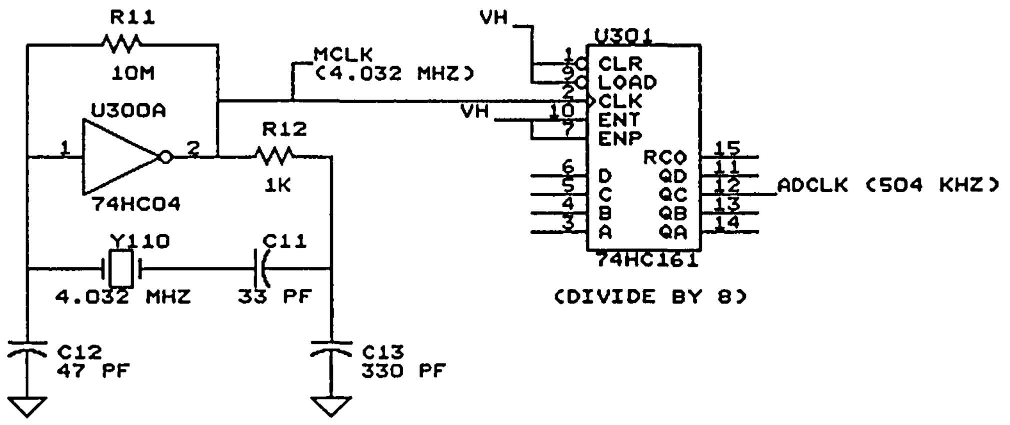


Figure 8.5 The 504 KHz square wave clock ADCLK for the A to D converter.

When the conversion process is over, an end of conversion (EOC) signal is generated and applied directly to the microcontroller as INT0 (INTerrupt 0). Since a polling rather than interrupt philosophy was adopted, an actual interrupt is not generated in the microcontroller but rather, at this time, the microcontroller is dedicated to monitoring the sense of this input. When the microcontroller senses that the conversion is complete, then it starts the data read process upon receiving an OE (Output Enable) signal made up of the combination in U400B (74HC02 NOR)

$$\mathrm{OE} = \mathrm{RD\ AND\ ADC.} \tag{8.3}$$

The A to D converter includes an eight channel multiplexer. The internal line COMP-IN (COMParison INput) is the input to the A to D section of the chip. An analog signal in the referenced range of 0 to +5 VDC will be converted to its eight bit digital equivalent. The internal line MUX-OUT (MUltipleXer OUTput) is the output of the eight channel multiplexer section of the chip. A given channel is selected through the channel address inputs ADDA through ADDD which are tied directly into the microcontroller as address lines A8 through A10 inclusive strobed by the START signal. These two lines, COMP-IN and MUX-OUT are tied together to connect the two sections of the chip.

The final circuit of the power module is the latch control circuit which is documented in Figure 8.6. The primary function of the power module is to control the distribution of +48 VDC power. This was done by a mechanical relay board that controlled 12 separate devices (including 10 string modules [seven optical modules, two calibration modules, one environmental module], the high speed communications laser (fiber optics driver), and the high speed circuitry of the string bottom controller. The programmable output pins $\underline{\text{INT1}}$ (INTerrupt 1) and T0 (Timer 0) were used to generate the chip enable signals $\underline{\text{LAT0}}$ and $\underline{\text{LAT1}}$ (LATch 0 and 1) respectively. These strobes are then used to latch in the eight parallel I/O lines IO0 through IO7 inclusive into either U409 or U410 (74HC573 or 574) eight bit wide latches (D flip-flop), the outputs of which are wired each to the control input of a single mechanical relay controlling the power to a single module.

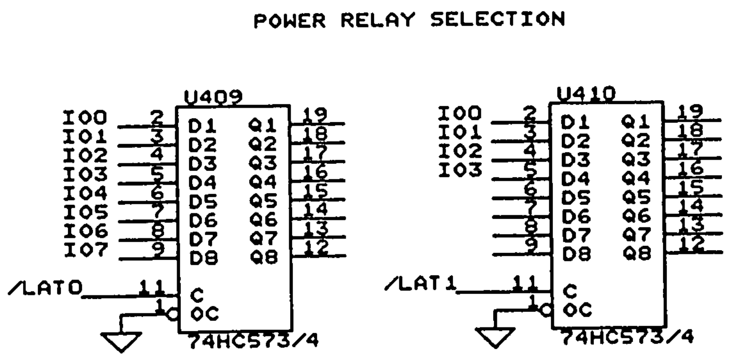


Figure 8.6 Power module latch control interface.

The default, upon power up, is to have all powers off. This is accomplished by having the microcontroller in the power module circuit, upon powering up, to enter a loop for several seconds issuing the commands that disable all of the power relays. Later, under operator command, each module may be turned on one by one.

8.3 Power Module Program Description

The power module program (PWR) is used to provide control and communications over the common party line string for power distribution module. Like its bigger brother, SOM, the PWR program spends most of its time "listening" to the string for commands. The principle objectives of this program are (1) to provide party line communications and (2) to provide control of the power distribution system of the string bottom controller.

The program PWR is made up of some 10 separate routines, not counting the modules associated directly with the UHPS language such as the telecommunications and timing routines. The routines found in PWR logically divide themselves into six classes as is summarized in Table 8.1 below:

Table 8.1

Summary of PWR Routines

Class	Name	Description
Main:		
	PWR	Main program
	INITIAL	Initialize stack and 300 baud serial port
Communications control:		
	ECHOM	Echo back valid command
	SYNCH	Synchronize on first byte of command string
Parser:		
	CMDPOLL	Parse the command string for a valid command
Command executive:		
	CMDSERV	Service the command request
Device drivers:		
	PWRON	Power on specified device
	PWROFF	Power off specified device
Error handling:		
	CMDERR	Report a command error
	FORERR	Report a format error

A complete description of this program would take up an inordinate amount of space. For the interested reader, Appendix 5 is a complete listing of this program including many in line comments. The program is written in the language UHPS which is detailed in Chapter 9. This chapter must be studied in order for the reader to be able to really follow the program listing in the detail necessary to make intelligent changes. What follows is a greatly simplified description of some of the salient features of the program. Understand that much of the important implementation detail is left out of this description so as to make the overall thrust of the program comprehensible.

The "main" or "root" level of PWR is flowcharted in Figure 8.7 below. Upon powering up, the microcontroller first executes an initialization routine INITIAL. This routine, which is common to both the optical module and power module programs, is flowcharted in Figure 7.10. It is initiates the system stack and sets up the bidirectional 300 baud serial communications port.

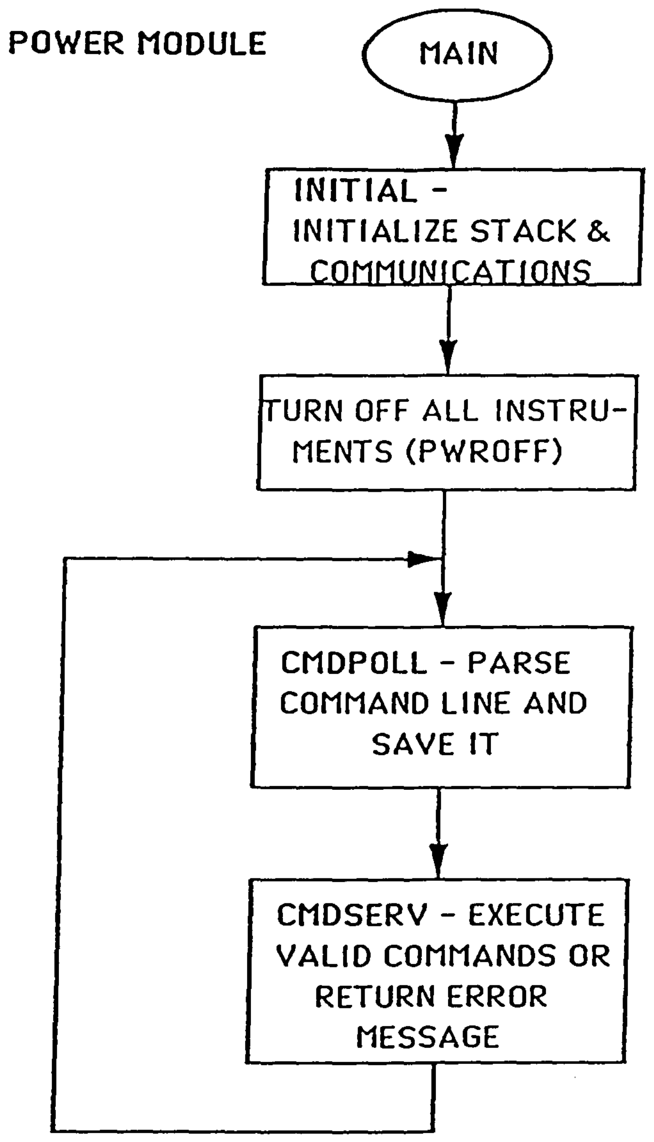

POWER MODULE
MAIN
INITIAL - INITIALIZE STACK & COMMUNICATIONS
TURN OFF ALL INSTRUMENTS (PWROFF)
CMDPOLL - PARSE COMMAND LINE AND SAVE IT
CMDSERV - EXECUTE VALID COMMANDS OR RETURN ERROR MESSAGE

Figure 8.7 Power Module main program.

After completing the initialization, the program first "gently" turns off one at a time all the devices whose power is controlled by the circuit by repeatedly calling PWROFF. Then, as a precaution, to be sure that indeed all the devices are off, it writes values directly to the latches snapping off any and all remaining devices all at once.

At this point, the program enters into an infinite loop. The function of this loop is to satisfy the main objectives of the program. The first objective, the handling of party line communications, is achieved in part by the routine SYNCH which is called from within CMDPOLL. This routine is common to both the optical module program and the power module program. All commands sent down to the power module are of the form:

$AA<CMD0><CMD1><CR> where

A	= device indentification number for the power distribution module
<CMD0>	= command byte 0 - either 'S' for toggling off the power or 'Q' for toggling on the power
<CMD1>	= hexidecimal power channel number (0..F)

Figure 8.8 Optical module command line

SYNCH is a busy wait loop which continues to look at successive characters in the serial communications stream for the special synchronization character '$'. CMDPOLL is the main parser routine, a highly simplified flowchart of it is given in Figure 8.9.

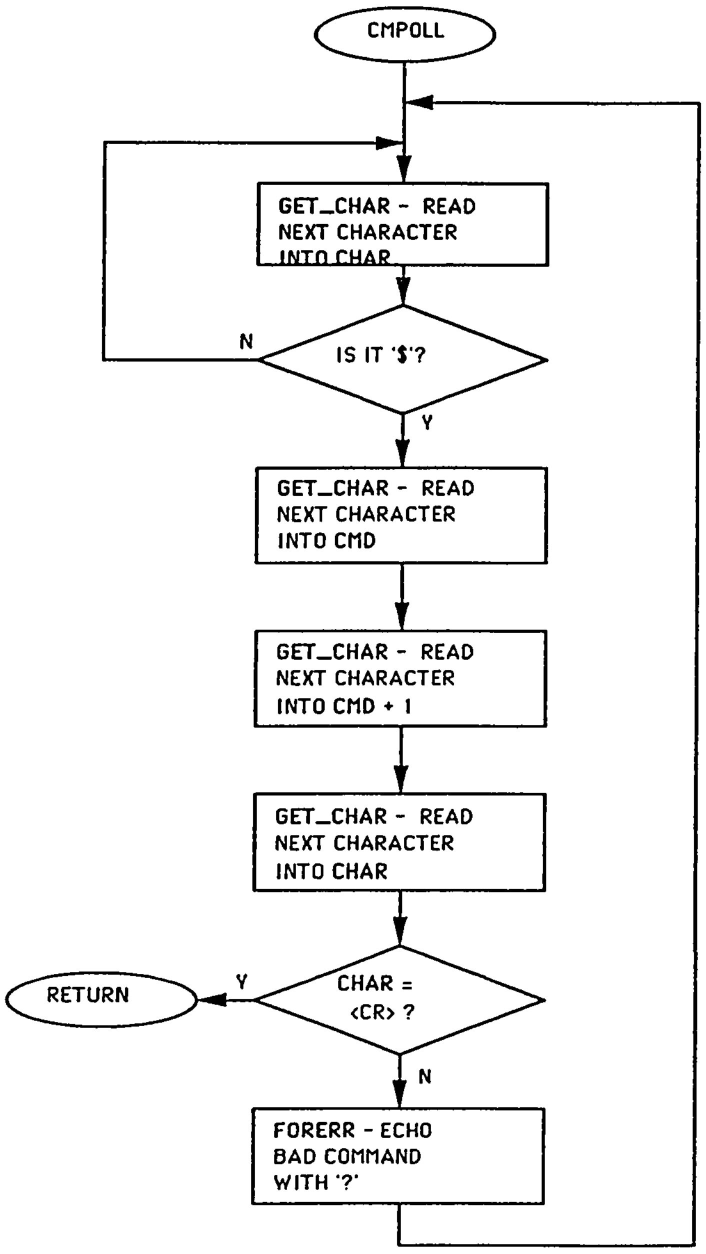

CMPOLL
GET_CHAR - READ NEXT CHARACTER INTO CHAR
IS IT '$'?
N
Y
GET_CHAR - READ NEXT CHARACTER INTO CMD
GET_CHAR - READ NEXT CHARACTER INTO CMD + 1
GET_CHAR - READ NEXT CHARACTER INTO CHAR
CHAR = <CR> ?
Y
RETURN
N
FORERR - ECHO BAD COMMAND WITH '?'

Figure 8.9 The main command line parsing routine CMDPOLL

Basically, CMDPOLL looks at each byte in turn after a synchronization byte is found. It parses the list of bytes to be sure that they adhere to the defined command line format above. If, in the course of parsing, a syntax error is encountered, the routine is exited with an format error message generated and ultimately PWR loops back searching for the beginning of the next command string. Finally, should a proper command for be encountered, CMPOLL is exited without any error conditions and PWR then calls CMDSERV.

Finally, Figure 8.10 flowcharts CMDSERV. This routine takes the command parsed out by CMDPOLL and checks to see if it is a lexically correct command. If it is not, a error reply message is generated by CMDERR. If it is correct, then the appropriate bit pattern is written into the latch , masking out all devices except the one to be toggled. This is done either by PWROFF or PWRON depending upon whether or not the device is to be powered off or powered on. A good reply message is generated by ECHOM and an error message is generated by CMDERR should a lexical error be encountered.

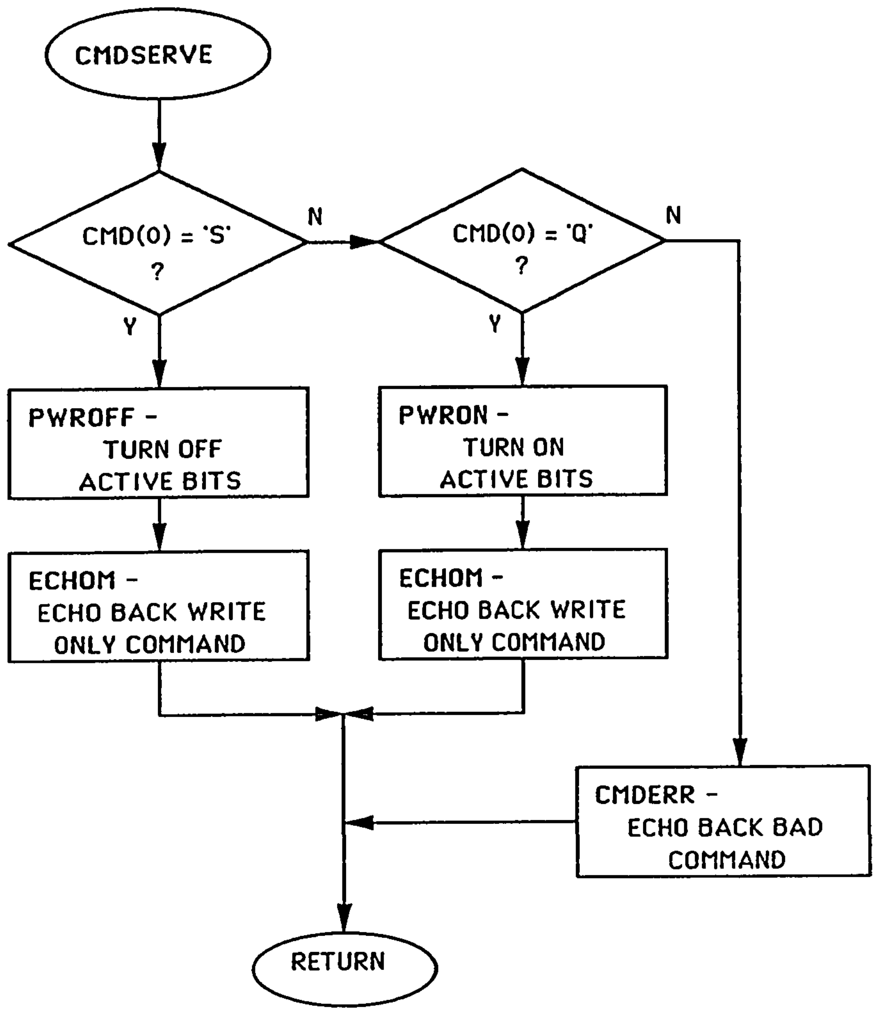

CMDSERVE
CMD(0) = 'S' ?
N
Y
CMD(0) = 'Q' ?
N
Y
PWROFF - TURN OFF ACTIVE BITS
PWRON - TURN ON ACTIVE BITS
ECHOM - ECHO BACK WRITE ONLY COMMAND
ECHOM - ECHO BACK WRITE ONLY COMMAND
CMDERR - ECHO BACK BAD COMMAND
RETURN

Figure 8.10 The command executive CMDSERV

CHAPTER 9 – Instrument Building Part IV – UHPS (Underwater Hawaii Programming System), a Structured Microcontroller Programming Language

9.1 Introduction

This chapter is a self-contained description of the motivation for and implementation of the Underwater Hawai'i Programming System (UHPS) which is a macro based structured programming language for the INTEL 8051 family of microcontrollers. As such, for those interested in the details of UHPS, its implementation and use, it is essential reading. The appendices containing the programs for the String Optical Module and the Power Module are written in this language. One may be able to gain some crude understanding of these programs without reading this chapter but the subties of implementation will be completely lost if this chapter is not throughly studied.

UHPS is a macro based language developed for programming the Intel 8051 family of microcontrollers. At the time programming for the optical modules in the SPS began, back in early 1984, there were no inexpensive, readilly available, programming languages for programming the 8051 microcontroller. An Intel MDS-225A microprocessor developement system with its associated ICE-51 in circuit emulator was purchased by the group to provide a direct hardware emulation environment for the 8051 microcontroller. Accompanying this system was the Intel (Microcontroller System) MCS-51 Macroassembler. The assembler had the contrasting characteristics of being very powerful in its pseudo instructions and macro options and yet the intrinsic instruction set was unbelievably crude.

The claim of the manufacturer is that the 8051 has 255 instructions (out of a possible 256 in an eight bit wide instruction set space). Further, they like to point out that 44% are intrinsic one byte instructions needing no operand and 41% need only a single byte operand implying a fast executing instruction set[1]. This is a very misleading claim.

To understand why, it is necessary to consider what a typical minimum microcomputer architecture might consist of. This idea has been around for quite a while and is today known by the term RISC for reduced instruction set computer.

To provide the reader with the proper background to understand the nature of the problem encountered with the very reduced instruction set provided by the 8051 microcontroller family, it is necessary to briefly introduce a few topics germane to the motivation and subsequent development of UHPS. First, an overview of computer architecture is presented. Then, a model minimum instruction set for a computer, the ABX model is discussed. Next, the actual architecture of the 8051 family is introduced which is compared to the ABX model. Then, the Intel MCS-51 cross assembler is briefly described in terms of its most important features along with a description of its macro facility MPL. Next, the initial motivation for UHPS is discussed in terms of filling in the missing operands needed by the 8051 to satisfy the ABX model. This was done by making extensive use of MPL. The subject of structured languages and their advantage of intrinsic modularity is considered. The implementation of program control structure in UHPS is described as a response to the dynamic requirements for code generation for the various 8051 based SPS modules since direct assembly level programming would have been a trap of uncontrollable spaghetti. Finally, the extensions of UHPS to overcome the very limiting internal memory limitations of the microcontroller along with a description of the use and implementation of primatives to hide from the application programmer many of the routine internal hardware peculiarities is given.

[1] The 8051 Users' Manual

9.2 Overview of Computer Architecture and the Von Neumann Model

The most commonly used computer architecture was developed shortly after World War II by the physicist Johnny Von Neumann and is known as the Von Neumann architecture. The basic idea of this architecture is ingenious in its simplicity. A memory array, i.e. a device capable of sequential storage and retrieval of quantized information, is used for two purposes: first, to store the instructions for the computer to execute and, second, to store the data that the computer works on. This was essential to do the artificial intelligence work that Von Neumann was interested in because it allowed the reinterpretation on an additional pass of the data as instructions thus giving the system a means by which it could "learn".

The Von Neumann architecture is not limited to artificial intelligence work but rather has been accepted as the most common architecture used in the computer world because of its inherent simplicity. Among other things, it makes system memory decisions flexible so that the memory can be allocated in different ways without having to change the hardware (i.e. if the program memory gets too large, one does not have to add program memory one simply reallocates software wise from the system memory).

A computer program is a list of sequential instructions that control the computer in such a way as to perform some meaningful task of information processing. These instructions come from a set of instructions intrinsic to the hardware of the system. Unfortunately, every computer has its own unique set of instructions and thus programs written on this level are not portable to other systems. Still, there are general guiding principles which do tend to transcend any particular machine. By orienting one's thinking in terms of some typical or common model, the task of programing on the instruction level can be simplified. This is particularly important in the present era of microcomputers in which the computer hardware itself is inexpensive and programming labor is expensive[2].

Probably the best known general purpose computer model is the "ABX" machine. The internal architecture and the instruction set constitute a very practical minimum system that all real systems ought to be a superset of. With this minimum system, it is possible to write any sequential computer program as long as the requirements do not exceed the intrinsic hardware limitations of the system (memory, speed, etc.).

2 Lewis, T. G.

9.3 The ABX Model

The ABX model is named after the three most basic internal registers of the model, namely the accumulator or arithmetic register (A) where the logic operations of the computer are performed, the basic auxiliry register (B), and the memory pointer or index register (X). Typical microcomputers process information in 8 bit (or one byte) wide pieces. They access this information (as well as fetch instructions to be executed) through a 16 bit (or two byte) wide memory array. Thus, the codes representing either instructions or data are limited to 256 (= 2^8) possible combinations and the memory space is limited to 65,536 possible memory locations (or addresses).

The internal architecture of a microcomputer ABX machine reflects this arrangement in terms of the registers found in the central processing unit (CPU) of the computer.

register A:	8 bits
register B:	8 bits
indeX:	16 bits
Program Counter:	16 bits
Stack Pointer:	16 bits
Condition Codes:	8 bits

Figure 9.1 The Registers of the ABX Programming Model.

Register A: 8 bit Accumulator register. The purpose of this register is to provide a connection between the arithmetic logic unit (ALU) of the computer which is the calculation logic of the computer and the computer's common memory. Data values read into this register along with absolute values or values referred to in memory, if needed, are acted upon by the ALU. The result is stored in the accumulator.

Register B: 8 bit Basic auxiliary register. The purpose of this register is act as a temporary internal storage area for data. Internal register operations are typically faster than memory references if for no other reason than that when a register is specified, its internal address is immediately known in comparision to the typical external address which must be calculated. Thus, when chain calculations are made, the ability to temporarily move a result from the A register into the B register so that an intermediate result may be found and then recall the previous result can enhance throughput.

Register X: 16 bit indeX register. The most primative use of this register is to act as a memory pointer or address register. Thus, this register is first loaded with an address and a subsequent memory to register transfer instruction is executed. Such an operation, of just holding the address, is sort of an address analog to the B register. In more advanced systems, the X register is capable of address arithmetic, analogous to the A register, and thus more dynamic methods of memory access are possible. This is considered in a bit more detail in the next section.

Register PC: 16 bit Program Counter. This register is the pointer to the memory location containing the current instruction being executed. Its counting is automatic in that when the instruction is fetched into the ALU, part of the ALU logic knows intrinsically how many bytes of operand to skip over to the next instruction. The major modification to this is a branch instruction in which the next instruction address is supplied as the operand of the branch instruction. Use of the PC is automatic and is of importance to the programmer mostly at debugging time when trying to single step through a piece of code.

Register SP: 16 bit Stack Pointer. A stack is a reserved area of general memory that is used as a sort of information scratch pad. It is a FILO (first in, last out) type buffer. For example, it is into the stack that context information (<A>, <B>, <X>, <PC>, and <CC>) is saved when a program is interrupted by calling a subroutine or a hardware interrupt is serviced. This register is used to point to the most recent entry of the stack memory.

Register CC: 8 bit Condition Codes. This register is made up of one bit flags indicating the status of various logic conditions. Typical of such flags may be the carry / borrow flag C (used in arithmetic overflow and underflow) and the zero flag Z (used to indicate if the most recent logic operation resulted in a zero).

The terminology and nomenclature for the ABX model is summarized below in Table 9.1:

Table 9.1

Nomenclature for the ABX model.

(a) Terms

accumulator	a register in which the result of an operation is formed
bit	a quantum (or unit) of information (either zero or one)
byte	a group of bits operated on as a unit (typically 8)
CPU	central processing unit, the collection of registers and arithmetic logic that makes up the central information engine of the computer
register	a device (typically internal to the CPU) capable of storing a specified amount of data (one or two bytes)
memory	a device (typically external to the CPU) capable of storing a specified amount of data (one byte)

(b) Operators

XY	= (internal memory) register or (external) memory XY
<XY>	= contents of XY
<--	= is transfered to
and	= Boolean AND
or	= Boolean (inclusive) OR
xor	= Exclusive OR
!	= Boolean NOT

Table 9.1 (Continued)

Nomenclature for the ABX model.

(c) Registers in the CPU

A	= register A (1 byte accumulator)
B	= register B (1 byte auxilary)
CC	= Condition Codes register (1 byte)
X	= indeX register (2 byte memory cell pointer)
XH	= high order byte of X
XL	= low order byte of X
PC	= Program Counter (2 byte program instruction pointer)
PCH	= high order byte of PC
PCL	= low order byte of PC
SP	= Stack Pointer (2 byte stack memory pointer)
SPH	= high order byte of SP
SPL	= low order byte of SP

(d) Memory and Addressing

M	= a memory location (one byte cell)
M +1	= the next higher location to M
Rel	= Relative address - number of bytes desired address is offset from a base address

Table 9.1 (Continued)

Nomenclature for the ABX model.

(e) Condition Codes register

C	= Carry / borrow flag - indicates if operation resulted in a carry (set) or borrow (clear)
Z	= Zero flag - indicates (set) if operation resulted in a zero value in the accumulator register (set) other bits are typically defined for this register such as controlling interrupts, etc.

Finally, it is instructive to establish a minimum instruction set for the ABX model. This is done in Table 9.2 where the instructions are divided into the catogories of general register and memory transfer, subroutine control, indexing, stack control, logic, and binary arithmetic.

Table 9.2

The operands of the ABX model

GENERAL REGISTER AND MEMORY TRANSFER:

LOAD ACCUMULATOR REGISTER <A>

LDA	LABEL	A <-- <M>
LDA	#$hh	A <-- #
TBA		A <-- <B>

LOAD AUXILARY REGISTER <B>

LDB	LABEL	B <-- <M>
LDB	#$hh	B <-- #
TAB		B <-- <A>

STORE MEMORY <M>

STA	LABEL	M <-- <A>
STA	LABEL	M <-- <B>

SET STACK POINTER FROM MEMORY <SP>

LDS	#immediate	SP(low) <-- <M>
		SP(high) <-- <M +1>

Table 9.2 (Continued)

The operands of the ABX model

SUBROUTINE CONTROL:

EXECUTE (AND RETURN FROM) SUBROUTINE

JSR	LABEL	push context on stack PC <-- new address
RTS		retrieve context on stack PC <-- old address

INDEXING:

LOAD MEMORY ADDRESS POINTER
(INDEX REGISTER) <X>

LDX	LABEL	X(low) <-- <M> X(high) <-- <M +1>

INCREMENT/DECREMENT ADDRESS POINTER

INX		X <-- <X> +1
DEX		X <-- <X> -1

Table 9.2 (Continued)

The operands of the ABX model

STACK CONTROL:

PLACE CONTEXT ON STACK MEMORY (SM)

PSH	A (excludes CC)	SM <-- <A>
		SP <-- <SP> +1

PSH	B	SM <-- <B>
		SP <-- <SP> +1

RETRIEVE CONTEXT FROM STACK

PUL	A	SP <-- <SP> -1
		A <-- <SM>

PUL	B	SP <-- <SP> -1
		B <-- <SM>

Table 9.2 (Continued)

The operands of the ABX model

LOGICAL:

COMPARISION

CBA		set CC by <A> - <B>
CMP	#$hh	set CC by <A> - #

BRANCHING

BRA (or JMP	LABEL LABEL)	PC <-- <PC> +2 +Rel

BEQ	LABEL	if <A> == <B> in CC
		PC <-- <PC> +2 +Rel

BNE	LABEL	if <A> != <B> in CC
		PC <-- <PC> +2 +Rel

BLT	LABEL	if <A> < <B> in CC
		PC <-- <PC> +2 +Rel

BGT	LABEL	if <A> > <B> in CC
		PC <-- <PC> +2 +Rel

BLE	LABEL	if <A> <= <B> in CC
		PC <-- <PC> +2 +Rel

BGT	LABEL	if <A> >= <B> in CC
		PC <-- <PC> +2 +Rel

Table 9.2 (Continued)

The operands of the ABX model

BINARY ARITHMETIC:

BUSY WAIT

NOP		PC <-- <PC> +1 (i.e. do nothing)

BYTE SIZE LOGIC

ADD	LABEL	A <-- <A> + <M>
ADDC	LABEL	A <-- <A> + <M> + <C>
SUB	LABEL	A <-- <A> - <M>
SUBB	LABEL	A <-- <A> - <M> - <C>
AND	LABEL	A <-- <A> and <M>
OR	LABEL	A <-- <A> or <M>
XOR	LABEL	A <-- <A> xor <M>
COM		A <-- !<A> == FF -<A> (one's complement)
NEG		A <-- -<A> == 00 -<A>
DEC		A <-- <A> -1

BIT SIZE LOGIC

LSLA	logical shift left all bits in A, clear A0	
LSRA	logical shift right all bits in A, clear A7	
RSLA	rotate shift all bits in A left thru C	
RSRA	rotate shift all bits in A right thru C	
CLC		C <-- 0
SEC		C <-- 1

9.4 The Index Register and the Stack

To more fully understand the ABX model, as well as the concept of structured programming which will be discussed below, two of the registers mentioned above, namely the index register and the stack need to be looked at in greater detail.

9.4a The Index Register and Different Methods of Memory Access

The index register is used in a variety of ways to point to a memory location. Some typical methods are as follows:

Immediate Addressing - a single byte operand which is the numerical value of the address being pointed to.

Relative Addressing - a single byte operand which indicates the offset (relative location) from the current location in the PC that must be either added or subtracted from the PC to access the desired location. Typically this is for a "nearby" jump instruction.

X := <PC> + <RA>

Indexed Addressing - address is the location contained in the index register.

X := <X>

Direct Addressing - single byte operand which contains the address in unsigned 8-bit binary form (thus intrinsically limited to the lower area of memory).

X := data-8

Extended Addressing - double byte operand which contains the address in unsigned 16-bit binary form.

X := data-16

9.4b The Stack

The stack in a microcomputer is a structure with associated operations of sufficient importance as to merit closer attention. The idea of a stack is to provide a reserved section of common memory that is used for temporarily holding information much as a scratch pad might be used. The organization of a stack is that it is a last in, first out buffer. As such, stacks are either push up stacks where successive information is placed sequentically in higher memory locations thus "burying" the earlier information or pull down stacks where successive information is placed sequentially in lower memory locations.

Access to the stack is made through reference to a hardware pointer, the stack pointer (SP) which always points to the current location, called the top of stack (TOS), where the next byte of information would be stored.

The advantage of a pull down stack is that you can place it in some used area of high memory and let it build downward while the program residing in low memory builds upward. One has to be a bit more careful in the case of push up stacks in allocating enough memory space for the stack.

Probably the most important use of a stack is in the storing and retrieving of context information in calling subroutines and handling system interrupt requests. Here the information required to return the system to the condition it was in before the interrupt is stored and then moved back into the system upon returning. The stack is also an excellent way of hiding information from different subroutines by passing subroutine parameters on the stack rather than allowing more than one subroutine direct access to a global variable (which if not carefully monitored can always be clobbered in some nasty way).

To not have a stack in a system is a nightmare. If one wants to temporarily store and retrieve some data without a stack, one has to explicitly define variables for every tempory storage of a variable and then one has to keep track of the storage on a case by case basis which is very prone to nasty errors. At best, one might have only as many variables set aside as would eventually be used by a stack for that purpose anyway and often many more. The stack solves all of this by keeping track of everything automatically.

The other use of a stack, namely calling subroutines, also is not absolutely essential in theory but is in practice. It is possible to write a computer program without subroutines but it rapidly becomes a very complicated affair. A subroutine allows one to break down a big problem into a succession of smaller problems. Often these smaller problems are needed more than once in the program and when written in subroutine form they do not require to be repeatedly written all over the program, just called. This technique is called modularization and is recognized as useful by every major computing language although to varying degrees.

Thus, the model computer architecture given here might be more correctly referred to as an ABX with stack model.

The use of a stack for passing parameters in subroutine calls has become increasingly important in the development of computer languages because of the concept of data (or information) hiding. In data hiding, a program is broken down into a number of subroutines each of which have access only to those variables that they really need to know about.

There are several advantages to this. First, it severly limits the number of possible interconnections that a subroutine has with the rest of the program. In practice, this means that the subroutine can be written and debugged independently of the program. Such a subroutine is called a filter which simply takes input information, transforms it, and then returns the transformed information as output. All that is needed to test a filter is to write a simple dummy driver program that passes values to a subroutine upon calling it and displays the returned values.

The disadvantage of this approach is that some throw away code, in the form of the driver, must be written. However, in practice this kind of code becomes very straightforward by simply setting up dummy input values, calling the routine, and then reporting the returned values. Thus, using known cases, one debugs the filter completely independently of the program.

From this point of view, a program then becomes basically a sequence of subroutine calls. In fact, the basis of the UNIX operating system is a mechanism by which the output of one filter is redirected to the input of another filter, etc. so that the command line becomes the actual program.

There are two basic methods by which one may use a stack to pass parameters to and from a subroutine. The simplest but less efficient method is to pass the parameters to the stack, then call the subroutine which in so doing will place the address of the return location from the subroutine call on the stack. Now, inside the subroutine, in order to make use of the information that has been passed, one must first pop off the stack the return address and temporarily save it. Next, one pops off the desired parameters. After completing the action of the subroutine one returns by carrying out the same steps in reverse.

Some computer architectures allow one to define any number of stacks in memory. With such an architecture, one uses one stack only for the purposes of passing parameters between subroutines and the other for handling the return addresses. In such a situation, execution speed is greater because there is no need to pop the return address off the stack in order to get to the passed parameters.

9.5 The Architecture of the 8051 Microcontroller Family

The architecture of the Intel 8051 series family microcontroller is plainly weird. To begin with, the chip has good hardware. As a microcontroller, in contrast to a microprocessor, the aim of the chip is to control a lot of hardware rather than to execute a lot of software intensive code. The connections of the microcontroller are designed in such a way as to make interfacing the chip with external hardware relatively easy. As such, there is a lot of hardware built into the chip.

Internal to the chip are a system oscillator which only needs an external crystal (and some by pass capacitors) to set the operating frequency, a restart circuit which only needs an external resistor and capacitor to define the restart timing, and four separate 8 bit wide ports which can be programmed down to the individual control line (bit). One of these ports has additional hardware connected to it which serves a number of very useful functions should one or more of these be desired in a given implementation. The single bit control lines in Port 3 can be used as follows:

P3.0 - Receive serial data
P3.1 - Transmit serial data
P3.2 - (hardware) Interrupt line 0
P3.3 - (hardware) Interrupt line 1
P3.4 - Timer 0
P3.5 - Timer 1
P3.6 - Write (external) memory strobe
P3.7 - Read (external) memory strobe

Figure 9.2 The single bit control line definitions of port 3 of the 8051 Microcontroller

Those lines which do not have to be dedicated to their built in hardware function are free to be used like the lines in the other ports to toggle hardware.

Port 0 also has a built in function and that is to act as the multiplexed data buss and lower address buss to any external memory. Similarly, additional lines of Port 2 are used as higher address lines. Finally, only Port 1 is completely free of any intrinsic function assignments and can only be used to control external hardware (i.e. as "I/O" lines).

External hardware is controlled quite simply in a microcontroller. In a traditional microprocessor, one assigns some address in the address space of the processor for the hardware to be interfaced at. Then, appropriate address decoding circuits must be implemented to provide the unambiguous address. In contrast, a microcontroller does not need any external address decoding circuitry. Instead, individual lines are directly accessible through the instruction set which can be used to toggle hardware on and off accordingly. Thus, there is no need to design an I/O area into the system memory.

In support of these intrinsic functions is an internal architecture of "special function registers" (Table 9.3) which are located at reserve areas of internal memory. Every intrinsic hardware function has one or more of these registers in support of it. This includes registers for buffering information, recording status, and programming control of the different hardware options. Additionally, among these registers are to be found the more usual CPU registers somewhat akin to the ABX model.

The memory architecture of the Intel 8051 family is unnecessarily clumsy. There are three different types of memory. Internal to the 8051 chip is what is called the "internal memory". It is a 4095 byte (FFF hex) long memory. The lowest 256 bytes (FF hex) of this memory is random access memory (RAM or read / write memory) with the upper 127 bytes (7F hex) of RAM being reserved as the locations of the special functions registers. Thus, the first thing that should be noted about the memory is that the CPU and hardware registers exist in it and are preassigned.

The 8051 family has two "external" memories. First, the external program memory which starts off where the internal program memory ended filling a memory space up to 65,536 bytes (FFFF hex) long. Second, the external data memory which starts at location 0 filling a memory space up to 65,536 bytes (FFFF hex) long. Thus, the second thing to note about the memory is that it does not conform to the Von Neumann architecture in which both code and data coexist! This kind of architecture, known as the Harvard architecture, precludes in a computer designed around it the sort of ideas that Von Neumann had of programs which could dynamically change their programming based upon some kind of feedback system. Ostensibly the advantage of such a system is that if the computer's program counter should somehow get confused and point to the wrong area of memory to fetch instructions, such an architecture precludes the possibility of the computer executing data assuming that it is code. The idea is that the computer may skip something but have a chance at recovery. This is really all quite "ify" since often in a hardware intensive environment if something is skipped a program could still hang waiting forever for a status bit to change which never will because something was skipped.

Table 9.3 The Special Functions Registers

8051 Family		ABX Model
ACC	accumulator	A
B	multiplication register	B
DPH	data pointer <high byte>	X
DPL	data pointer <low byte>	X
IE	interrupt enable	
IP	interrupt priority	
P0	port 0	
P1	port 1	
P2	port 2	
P3	port 3	
PSW	program status word	
SBUF	serial port buffer	
SCON	serial port control	
SP	stack pointer	SP
TCON	timer control	
TH0	timer 0 <high byte>	
TH1	timer 1 <high byte>	
TL0	timer 0 <low byte>	
TL1	timer 1 <low byte>	
TMOD	timer mode	

The instruction set of the 8051 and its capabilities are another matter altogether. To quote the Intel literature[3] "The MCS-51 instruction set includes 111 instructions, 49 of which are single-byte, 45 two-byte and 17 three byte." The implication of all of this is that the microcontroller is fast. This is because of the large number of instructions requiring only one or two bytes of code and thus the machine should not be having to expend too many cycles on instruction fetching but rather on instruction execution.

[3] The 8051 Users' Manual, p. 8-1

Further, when one looks into the instruction set just a little bit further, one finds that the 111 instructions are implemented in 255 (out of a theoretically possible 256 for one byte wide instructions) separate opcodes. Why is the possible opcode space so full?

There are a number of reasons why the opcode space for the 8051 family is so crowded. The biggest one is its designer's obsession with so-called fast, one byte instructions. For example, there are 32 possible I/O lines that can be independently toggled (assuming that that is all you want to do and that you do not want to talk with external memory or handle interrupts or serial communications or make use of any of the other built in hardware in the microcontroller). Every one of these 32 lines has associated with it a separate instruction for toggling on or off the bit associated with the I/O line. Thus, already, 64 of the 256 possible instructions are accounted for. This philosophy prevails throughout the instruction set so that it is very full of instructions that do not do very much. Other microcontrollers and microcomputers sacrifice speed of the individual instruction to have a second byte or operand that in this case would point out to only two separate instructions, namely the instructions to either set or clear the I/O line bit, which I/O line was intended for the operation. As such, two bytes, namely the instruction and the operand would have to be fetched, but the opcode space would have considerably more space left in it for more comprehensive instructions which, if intelligently chosen and used, could significantly speed up overall system throughput. In the case of the 8051 family, the opcode is so full of truely primitive instructions that there is no room left over for what within the ABX model or its like would be considered barely essential.

Another limitation of the 8051 is the use of the lowest 32 bytes of its internal data memory. It has the flexibility to be programmed a number of ways which on the surface may seem like an advantage but which in practice can be a real pain. The lowest 32 bytes can be divided into one to four "register banks" consisting each of eight registers each labeled R0 through R7 inclusive. These are general purpose (data) registers like the B register of the ABX model. Also, R0 and R1 can be used as eight bit wide pointers. Basically, the idea is that if one wanted to speed things up by not having to save the context on a stack everytime an interrupt or a subroutine changed the execution path of the program, the context is automatically saved by switching register banks. One of the problems with this approach is that one is limited to subroutine calling only four layers deep (something not necessarily true in the optical module). Actually, the main problem comes about that now the programmer must keep track of just how deep subroutine calls are now at any given time in the execution of the program rather than letting the system handle it. Additionally, there are plenty of cases in which the context is not a whole eight bytes wide and as a consequence there is a wasting of very limited internal memory. Instead, only one register bank was used (bytes one through eight) and a stack was implemented in the remaining space (bytes nine through 32 which runs right up against the reserved area for the internal hardware registers). The remaining internal data memory not used by the register bank, stack, or hardware was used as general purpose memory for holding variables, etc. Altogether, the internal data memory, including reserved hardware registers, consists of only 128 bytes.

Another misleading feature of the 8051 family is the 64 kilobytes of "code" memory. Part of this can be internal in the sense that the 8751 part has a built in 4 kilobyte ROM. If the code exceeds this, then all of the code must be in an external ROM since the control line that indicates that the memory requirement is greater also disables the internal ROM. The biggest problem is that this is indeed a separate memory from the data memory (following the Harvard architecture rather than the Von Neumann architecture) and there are absolutely no instructions permitting any kind of data transfer to and from the code memory and the ALU of the microcontroller. Thus, ideas such as having the ability to override a default program by down loading a new program are impossible to implement. Fortunately, for the optical modules and other devices in the SPS using the 8051 family, this does not seem to be a problem (the ability to override a program existed with the SB-180 computer used in the SBC). This inability to download a program made testing the system potentially very difficult. The ICE-51 in circuit emulator was specifically obtained to avoid the dangers of being forced into having to burn a ROM every time a program change was made and needed to be tested.

Finally, it should be pointed out that one of the truely desirable goals of any instruction set designer is to develop a symmetrical instruction set. What this means is that operations generally have an inverse operation. If there is an addition, there should be an accompanying subtraction, etc. This is not the case at all for the 8051 family instruction set. An annoying example of this is the case of the datapointer incrementation instruction. It exists, but the datapointer decrementation instruction does not!

9.6 Analysis of the 8051 Instruction Set

It is instructive to see how truely deficient the 8051 family instruction set is by directly comparing it to the ABX minimum model. There are certainly no complaints about the instructions that are peculiar to the direct control of the internal hardware of the 8051 since that is what it is advertised to do better than other microcontrollers. However, for a comprehensive control program to be written, say beyond some simple look up table such as a keyboard encoder circuit, the minimum ABX model or its equivalent should be satisfied or a lot of time is going to be spent fighting with the instruction set to hammer something out. This comparison is done in Table 9.4 which reproduces the opcodes of the ABX model defined in Table 9.2.

Table 9.4

Comparison of the operands in the 8051 family to the ABX model

ABX OPERAND	8051 OPERAND
GENERAL REGISTER AND MEMORY TRANSFER:	
LOAD ACCUMULATOR REGISTER <A>	
LDA LABEL	MOV A, LABEL
LDA #$hh	MOV A, #hh
TBA	
LOAD AUXILARY REGISTER <B>	
LDB LABEL	MOV B, LABEL
LDB #$hh	MOV B, #hh
TAB	
STORE MEMORY <M>	
STA LABEL	MOV LABEL, A
STB LABEL	MOV LABEL, B
SET STACK POINTER FROM MEMORY <SP>	
LDS #$hhhh	

Table 9.4 (Continued)

Comparison of the operands in the 8051 family to the ABX model

ABX OPERAND	8051 OPERAND
SUBROUTINE CONTROL:	
EXECUTE (AND RETURN FROM) SUBROUTINE	
JSR LABEL	CALL LABEL
RTS	RET
INDEXING:	
LOAD MEMORY ADDRESS POINTER (INDEX REGISTER) <X>	
LDX LABEL	MOV DPTR, #hhhh
INCREMENT/DECREMENT ADDRESS POINTER	
INX	INC DPTR
DEX	

Table 9.4 (Continued)

Comparison of the operands in the 8051 family to the ABX model

ABX OPERAND		8051 OPERAND
STACK CONTROL:		
PLACE CONTEXT ON STACK MEMORY (SM)		
PSH A		PUSH LABEL
PSH B		
RETRIEVE CONTEXT FROM STACK		
PUL A		POP LABEL
PUL B		
LOGICAL:		
COMPARISION		
CBA		
CMP	#$hh	
BRANCHING		
BRA	LABEL	
BEQ	LABEL	
BNE	LABEL	
BLT	LABEL	
BGT	LABEL	
BLE	LABEL	
BGT	LABEL	

Table 9.4 (Continued)

Comparison of the operands in the 8051 family to the ABX model

ABX OPERAND		8051 OPERAND	
BINARY ARITHMETIC:			
BUSY WAIT			
NOP	NOP		
BYTE SIZE LOGIC			
ADD	LABEL	ADD	A, #hh
ADDC	LABEL	ADC	A, #hh
SUB	LABEL	SUB	A, #hh
SUBB	LABEL	SBB	A, #hh
AND	LABEL	ANL	A, #hh
OR	LABEL	ORL	A, #hh
XOR	LABEL		
COM			
NEG			
DEC			
INC	INC LABEL		

Table 9.4 (Continued)

Comparison of the operands in the 8051 family to the ABX model

ABX OPERAND	8051 OPERAND	
	BIT SIZE LOGIC	
LSLA	RL	A
LSRA	RR	A
RSLA	RLC	A
RSRA	RRC	A
CLC	CLR	C
SEC	SETB	C

Scanning Table 9.4 shows that the major deficiencies in the 8051 instruction set when compared to the ABX minimum model lie in the area of comparison and branching instructions. Believe it or not, there are no simple comparison instructions in the 8051 instruction set by which a decision might be made in alternative program pathways! One composite (definitely not simple) instruction does exist. This instruction is the "compare, jump not equal" instruction written symbolically as:

```
CJNE        A, #hh, LABEL
```

where the LABEL is the destination of the program should the condition that the contents of the <A> register are not equal to the immediate value hh. Since the value being compared is indeed an immediate or absolute value, it is a constant that must be determined at assembly time and not something that can be computed during program execution. This is a very severe restriction on program control.

One "branch" instruction also exists which again is a composite instruction and of limited use. This is the "decrement and jump not zero" instruction written symbolically as:

```
DJNZ        LABEL1, LABEL2
```

where LABEL1 is the destination should the <A> register be cleared and LABEL2 is the entry point for the loop. This is a fairly sophisticated instruction, the kind one finds in more powerful instruction sets than the ABX model where the desire is to help the development of structured programming languages for the computer. However, one instruction of this kind does not make up for the lack of the other similar instructions that would be needed for an advanced system nor does it make up for the lack of minimal instructions demanded by the ABX minimum architecture.

As has been pointed out, the 8051 is very limited in terms of its available "internal data" memory. In developing the program for the optical module, this limit was exceeded. It turns out that if one is to deal with "external data" memory, then the instruction set is even further reduced to only two instructions, namely the ability to transfer data between the internal and external data memories by way of the <A> register written symbolically as:

```
MOVX      @DPTR, A
MOVX      A, @DPTR
```

Finally, in terms of major deficiencies in the 8051 instruction set, it must be noted that the logic and arithmetic functions are again limited because they compare only the <A> register to an immediate value. This is not quite as bad a deficiency since at least the operations exist. Interestingly enough, the 8051 also includes instructions for multiplication and division, again, beyond the minimal ABX model.

In sum, the designer of the 8051 instruction set seems to not be able to make up its mind what it is. On the one hand, it has useful internal hardware with appropriate control instructions that make it attractive. It has some particularly advanced instructions, yet these are more of a tease than useful since they are quite incomplete. It is capable of being set up in a number of different modes, both stack oriented and register bank oriented, yet its usable "internal data" space is very limited indeed. As such, it almost assures that "external data" memory will have to be accessed, yet it provides a very limited set of instructions for this. Thus any use of external data memory is bottle necked by having to pass everything to and from external data memory to internal data memory. Additionally, with only one accumulator (the <A> register), as is the fashion of INTEL instruction sets, everything is bottlenecked going in and out of the accumulator too. It is the single accumulator that is the first thing that is always criticized in the ABX model. Comparison with the ABX minimum model shows that the 8051 has no useful comparison or branching instructions making anything other than the simplest of programs a real nasty chore to write let alone debug or modify.

It is precisely these observations that form the motivation for developing the programming language which is the subject of this chapter.

9.7 A Look at the Conventions of the MCS-51 Cross Assembler

MCS-51 is the cross assembler supplied by INTEL in support of its 8051 family of microcontrollers. It runs on the INTEL MDS-225A Microprocessor Development System and its assembled code is downloaded into the circuit under test through the INTEL ICE-51 In Circuit Emulator. Since it supports the 8051 instruction set, in this respect, the assembler is quite crude. On the other hand, it has imbedded in it a tremendous number of functions, the most important of which will be covered here. These functions include preassembler commands, pseudo-operands, link referencing, memory segmentation, numerical resolution, and in line comments among others.

A quick way to get into MCS-51 with an eye towards understanding UHPS is to look at some of the conventions of MCS-51. These are briefly listed in the following:

Names - there are several kinds of naming conventions used in MCS-51. First, the file name is dictated by the limitations of the ISIS-II operating system that runs on the MDS-225A. Filenames are made up of a maximum of six alphanumeric characters in a string with the first character always being alphabetic followed by the option of a delimiting period and up to three alphanumeric characters as an extention. Thus, a representative file name may have the form:

ISITHX.A51

Each file has internal file name. This is indicated by directly preceding keyword "NAME". This name must be an alphanumeric string of (essentially) any length. All alphanumeric strings in MCS-51 may use an underline character to connect together words to form a continuous string. Strings are terminated by white space. Thus, near the begining of the file, one may label the file internally as:

NAME IS_IT_HEX

Labels within the code of an MCS-51 file, used to designate a point within the program (code memory), are terminated with a colon as in:

CODE_LOCATION:

Strings used to label variables and constants abide by the same rules but have no terminating colon.

Comments - can be placed anywhere within the file. Comments are indicated by a single semi-colon character being placed before whatever is to be commented on that line. The only restriction is that once a comment is started on a line, everything which follows it on that line, with the exception of a macro declaration (see below), will be interpreted as a comment. Typically, one uses comments in three ways. One, to place comment lines between code lines. Two, to place a comment to the right of any code in a line (following the usual assembler convention of LABEL OPCODE OPERAND COMMENT). Three, to "comment out" any line of code that one might wish to temporarily remove.

Preassembler commands - these are commands to the MCS-51 assembler to do special things when they are encountered. They are indicated by a leading dollar sign character before the command. Two commands are of particular importance. First, the command

$DEBUG

at the very top of a file indicates to the assembler to retain code memory symbols of that file in a special symbol table that is created by the relocatable linker (RL-51) and retained and loaded by ICE-51 during programming testing. As such, break points can be set using those symbols (assuming that they are not overridden by the tendency for ICE-51 to mix up all symbols, from all memories, and to only retain the last one with the same value greatly confusing everything especially since INTEL went out of its way to separate everything with the use of the Harvard architecture).

The second command,

```
$INCLUDE (FILENM.EXT)
```

calls for the assembler to locate and read in the file named FILENM.EXT. This is real handy for calling in definitions of constants and macros as will be seen later.

Link references - are made explicitly in MCS-51. If no explicit reference is made, then all routines (code addresses) are automatically local and not knowable by other routines. This is good modular programming technique. Two commands cover this. First, the command to declare that the routine defined in the current file is available for linking with others is

```
PUBLIC      LABEL1, LABEL2, LABEL3
```

Thus, in this case, the routines (or entry points) LABEL1, LABEL2, AND LABEL3 are made available. The second command is

```
EXTRN (SUBR1, SUBR2, SUBR3)
```

This indicates that the routines SUBR1, SUBR2, and SUBR3 are needed by the code in the current file and that they are to be found elsewhere. Thus, the linker RL-51 uses these two commands to identify and link up routines. What this means, is that the same symbols may be used in more than one file as long as none of them are declared as "global" with the above commands.

Memory segmentation - is explicitly indicated in MCS-51 by the use of the commands SEGMENT and RSEG being associated with one of the three possible types of memory, namely DATA (internal data), CODE (program), or XDATA (external data). The first command is used to associate a name for the memory segment as well as to define the type of segment it is. For example,

```
ATOD_DATA    SEGMENT  DATA
```

defines the memory segment as a DATA segment and assigns the name ATOD_DATA to it. Later, just at the point where one wishes to define the contents of the segment either by filling the space with something such as code or constants or by reserving space for variables, one uses the second command as in

```
RSEG         ATOD_DATA
```

This will continue to be the space that everything following will be put into until a new RSEG command is given with a differently defined and named space. The obvious exception to this is if code in the form of operations and operands is given. Then the assembler unambiguously knows that this must go into the most recently defined code space.

Originating memory segments - is accomplished by a couple of pseudo-operations. Code segments begin with the pseudo-operation

```
ORG     OH
```

here indicating that the segment began at location OH. Data segments are originated with

```
DSEG  at  8
```

for an internal data segment that begins at location 8 (i.e. the byte just beyond the first possible register bank). Finally, external data memory segments are originated with

```
XDATA    100H
```

for an external data segment that begins at location 100H (i.e. the byte just beyond the internal memory space).

Numeric representations - a series of conventions, closely akin to most other commonly found assemblers, are used to represent informational patterns on a byte by byte basis. Numbers can be represented in one of three ways which are demonstrated here for the number eight:

8	decimal
00001000B	binary
08H	hexadecimal

In the case of immediate addressing, etc. where one is going to use a number directly as a constant value as an operand in an instruction, the '#' symbol is placed in front of the number. Finally, in this regard, ASCII strings can be directly represented by enclosing them in single quotation marks such as in:

'This is an ASCII string.'

Strings - are usually one of two kinds. Code strings, by their very nature, are fixed in length. As such, two pieces of information are used to control them, the address of the first entry into the string and the length of the string in bytes. Data strings, being variable, require a special terminator character, such as the value 0H. Again, the address of the first entry into the string is supplied and the string is "gobbled up" byte by byte until the terminator character is encountered.

Pseudo-operations - are commands that are entered into the assembly code as if they belonged to the instruction set of the microcontroller and yet do not. These commands inform the assembler of choices and other information that it needs to translate the list of assembly mnemonics which make up the program into code that the computer can directly execute. Several have been pointed out above. The remaining important ones are now considered. Basically, one wishes to either reserve space in data memory for variables or to define values of constants in code memory. Variables in data memory are defined by the "define space" pseudo-operation as in:

```
STACK:      DS    24      ; reserve space for a stack
```

Constants in code memory are defined by the "define byte" pseudo-operation as in:

```
STRING:     DB            'This is a string'
```

Finally, along these lines, one can assign names to values and to other names. The most common one is the "equate" pseudo-operation as in:

```
JOHN            EQU         FUNG_CHU
FUNG_CHU        EQU         69
```

so that different names can be used to reference the same value. This is very important to use when the same value is being used in many different parts of a program for should the value be changed, the proper use of EQU can save a lot of heartache in having to properly track down all occurrences to change them.

Single bits can also be defined similarly. In the 8051, more typically, one toggles on and off the single I/O bits which can be directly addressed by their reserved names such as P3.2 for the second bit in port three which you might associate in your application as say the "hallway light". In this case, you would define

```
HALL_LITE       BIT         P3.2
```

9.8 The Concept of a Macro Assembler and a Summary of the MCS-51 Macro Facility MPL

An assembler is a computer program which translates the mnemonics representing the different computer instructions into the numerical code of those instructions. Additionally, at assembly time, it calculates referenced addresses such as a branch to a subroutine. This saves an enormous amount of time by not having to depend upon programmers who would otherwise have to spend most of their time hand translating the code.

A macro instruction is a single instruction name of a list or a sequence of other instructions. A subroutine is a sequence of instructions that appears only once in a program while it may be called (referenced) from many different places. A macro differs from a subroutine in that everytime the macro name appears, the macro handler in the assembler expands the macro into its component instruction list. Thus, the macro may appear in more than one place in the program and it is never referenced but rather expanded when encountered. This expansion is done at assembly time and is an exercise in string substitution (substituting the instruction list for the macro name). A good macro handler permits the use of dummy string variables. This makes the macro a powerful tool allowing variations on the common theme of the macro. Typically a macro is a small piece of code which may be viewed as a single instruction with options. The options are the actual parameters passed to the macro at the time it is expanded. A subroutine on the other hand usually is more like a function that does some complete job in and of itself and so tends to contain much more code than the typical macro.

The variables in a macro are fixed at assembly time whereas the variables in a subroutine are fixed only at the time the subroutine is called. As such, the variables in a macro are quite static compared to those in a subroutine.

For example, let us say that there is no instruction in the instruction set for transfering a byte stored at one location in memory to another location in memory. One only has the ability to load a byte of information from a memory location (LDA loc1) to internal register A and store it back again to another memory location (STA loc2). Should it turn out that the nature of one's program is such that the need to make memory to memory transfers occurs quite often, then this is a prime candidate for defining and invoking a macro instruction. Such a macro might be defined as:

```
%*DEFINE(MMT(X,Y)) LOCAL LABEL
(
        LDA  %X
        STA  %Y
)
```

where X and Y are the macro dummy variables. The macro instruction name is MMT (memory-memory transfer). Macros are typically collected together in an include file so that they are easilly modified by the programmer while made readilly available to the assembler. Let us say that later on in the program, there are two memory locations called PORT23 and INBUF2. The need arises to move the byte contained in PORT23 to INBUF2. Rather than explicitely write out the code to do this in the intrinsic instructions of the assembler, one can invoke the macro MMT by the line

```
%MMT(PORT23,INBUF2)
```

which then tells the macro handler of the assembler to expand this code and to substitute PORT23 for the dummy variable X and INBUF2 for the dummy variable Y.

9.9 Filling out the 8051 Instruction Set -- The Origins of UHPS -- the Runtime Macros (RUNMAC.INC)

As was pointed out above, in comparing the 8051 family instruction set to the mimimal ABX model, the most glaring deficiencies are in the areas of comparing and branching instructions. There are none! This is the original motivation for the development of UHPS.

In order to be able to do basic decisions in program control, both comparison instructions for different logical combinations and branching instructions are needed. This lack was made up for by a defining and using a series of macros and associated primitive subroutines (called "primitives" for short, these are explained in more detail later on) that filled in for the missing instructions.

The primitives are all found in the file RUNLIB.P51. Altogether, there are six similar primitives which compare two variables, X and Y, and return either a TRUE or FALSE condition depending upon the logical comparison requested. The six primitives handle the six logical comparisons as shown in Table 9.5 below:

Table 9.5 The six logical comparison primitives

Logic function	Primitive
X> Y	XGTY
X< Y	XLTY
X= Y	XEQY
X!=Y	XNEY
X>=Y	XGEY
X<=Y	XLEY

It is illustrative to see how this is accomplished. As an example, consider Figure 9.3 which shows the XGTY primitive:

```
XGTY:

          MOV     A,R1     ; Y
          CPL     A
          MOV     R1,A     ; !Y
          MOV     A,R0     ; X
          CLR     C
          ADDC    A,R1
          RET
```

Figure 9.3 The comparison primitive XGTY

Upon calling XGTY, the value of X is moved into <R0> and the value of Y is moved into <R1>. This is a common programming model for all six comparison primitives. The value of Y is then complemented to produce its binary additive inverse. Next, the value of X is moved into the accumulator <A> and the <C> (carry) flag cleared. Finally, the complemented Y and the X values are added with the <C> flag being set if and only if a carry is needed, i.e. X is not greater than Y. The result is returned as the carry flag to be read by the calling routine.

As a reminder, consider the equivalent code in the ABX model, namely

```
LDA
LDB
BGT  ELSEWHERE
```

where the comparison and the branch have all been accomplished in three instructions. We have yet to do the branching.

Similarly, one calls such primitives with code like the following:

```
MOV     R0,XVALUE
MOV     R1,YVALUE
CALL    XGTY
```

and then one proceeds to test for the result in the <C> flag. The point is that the designers of the 8051, in the enthusiasm to develop a microcontroller with fast executing instructions, force the programmer to use many more instructions than would otherwise be used wasting expensive programmer's time[4] and memory space all with no real net increase in throughput.

To facilitate the incessant calling of these comparison primitives, it makes sense to "encapsulate" them within macros that are much closer to what the programmer really wants to do, namely make a direct comparison between two values and then branch accordingly. The file RUNMAC.INC contains, among other things, six complementary calling macros as shown in Table 9.6.

Table 9.6

The six complementary calling macros for the comparison primitives

Primitive	Complementary macro
XGTY	MXGTY
XLTY	MXLTY
XEQY	MXEQY
XNEY	MXNEY
XGEY	MXGEY
XLEY	MXLEY

[4] Lewis, T. G. (1979)

MXGTY is a typical example which is defined in Figure 9.4:

```
%*DEFINE(MXGTY(X,Y)) LOCAL LABEL
(
        MOV         R0,%X
        MOV         R1,%Y
        CALL        XGTY
)
```

Figure 9.4 The MXGTY calling macro

so that in our code now, the single call

```
MXGTY(XVALUE,YVALUE)
```

does it all! One still needs to evaluate the returned logic value in the <C> flag and then make the branch, but this is a lot closer to what one would want. Before going any further, it is useful to take a look at the subject of branching in computer programs.

9.10 The Conventions of UHPS

The conventions of UHPS are an extension of those of MCS-51. The overridding philosophy was to adopt a set of conventions along the line of the programming language PASCAL. PASCAL was chosen as a model because in it everything has a place and this kind of imposed discipline was needed to tame MCS-51 so that a consistent, relatively easy to modify and test, programming system would evolve.

PASCAL is famous for a "formula" which in modified form could be expressed like this

Module = Data Structures + Logic Control Structures
+ Algorithms[5]

The term module refers to the idea that a program is logically broken down into one or more subprograms (subroutines or functions). Typically, there is one such module per file and so isolated, the module is debugged independent of its application. Eventually, all of the modules are linked together to produce the desired program. Some code in the form of "throw away" test code must be written, but this kind of code becomes very easy to write and this kind of emphasis on dividing and conquering the programming task tends to result in more general purpose modules which get used over and over again in different programs so that time and effort are not wasted reinventing the wheel. Finally, in this regard, such modularity permits the development of large software projects with more than one programmer since once the required modules are identified, different programmers can be given the task of writing different ones as long as agreement is reached in terms of information passing.

[5] Jensen, Kathleen and Wirth, Niklaus (1974)

The initial conventions of UHPS to be considered here are basically those in the category of presenting data structures. A PASCAL like order of doing this thus divides a given module into two separate parts, a "header" section (the data structures) and a "code" section.

The header section is started with the internal file name following the MCS-51 conventions mentioned before. For UHPS this is typically a "long" version of the file name chosen to represent what the function of that module is. Next, comes the basic order of information listing as follows:

Description of the module - this is a series of comment lines identifying the purpose of the module, how information is passed into and out of it, and any information concerning who wrote it and what the latest revision date is.

Link references - here the global declarations are made using the MCS-51 commands PUBLIC and EXTRN as described before.

Global variables and constants - are defined next using the MCS-51 pseudo-operations DS, EQU, and DB. Those so defined here are declared immediately above by the PUBLIC command so that the linker RL-51 can find them. Typically, a set of common constant definitions, which is merely a list of EQU statements, is called in from an external file using the $INCLUDE preassembler command.

Local variables and constants - are defined next, again using the MCS-51 pseudo-operations DS, EQU, and DB. Since they are not explicitly declared global, they are local to the module.

Data memory segmentation - here the data memory segments, both internal (DATA) and external (XDATA) are declared and assigned names using the MCS-51 SEGMENT and RSEG commands as described before.

This sort of an arrangement is always better documented by an example. As such, Figure 9.5 summarizes a typical header section for a file named EXAMPL.A51:

```
NAME EXAMPLE_MODULE

; subroutine EXAMPLE - this subroutine is an
;example module that does nothing but shows
; the imposed order of UHPS on to MCS-51

; author - Name Of Author, University of ...
; revision date - June 30, 1997

; link references

        PUBLIC    EXAMPLE     ; this module is the
                              ; source of EXAMPLE
        EXTRN   CODE (XEQY)   ; and needs the
                              ; external routine
                              ; XEQY

; global constants and variables

$INCLUDE  (CONSNT.INC)        ; which is a big
                              ; collection of EQU
                              ; statements
```

Figure 9.5 An example header file named EXAMPL.P51 in UHPS

```
; local definitions and variables
;
; const

        TSWITCH   BIT  P2.4 ; a local constant

XAMP_MSG  SEGMENT CODE
RSEG XAMP_MSG

        XAMP_INSTO:
                DB 'This is a dummy message',00H

; var

XAMP_DATA SEGMENT DATA
RSEG XAMP_DATA

OCHAN:   DS   1    ; output channel

; begin subroutine
```

Figure 9.5 (Continued) An example header file named EXAMPL.P51 in UHPS

This example points out several things. The include file CONSNT.INC is a collection of universal constants that is almost always included in every UHPS module. It is a list of EQU statements and is discussed later in detail. The terminating comment line

```
; begin subroutine
```

is the boundary between the heading and code sections of the module. Finally, note that the names given to the different memory sections found in the module are all variations on the theme of the module name (either the external short file name or the internal long name). This helps identifying where things come from in reading over a symbol table generated by the relocatable linker RL-51.

In practice, as UHPS evolved, CONSNT.INC is not the only include file (and thus the reason for the extention name ".INC") commonly used. The two others, which will be looked at in detail later, are RUNMAC.INC and LCSTRC.INC which contain the run time macros and logical control structures of the language respectively. So many files in fact required all three of these that the file reference UHPS.INC is often included instead. It is merely a list of these three common include files.

Next, let us consider briefly the conventions of the code section. Basically the section is begun with an address label that is based upon the name of the module, in our case EXAMPLE: would do fine. This is followed by the code in the form of 8051 operations and operands, UHPS macros, and MCS-51 pseudo-operations. Whenever intermediate address labels are needed, these are based upon the starting address label as a sort of variation on the theme. In our case these may be something like EXAMPLE1: and EXAMPLE2:. Finally, the code is terminated by defining a final address label at the very end of the code. By convention in UHPS, this is the starting address label spelled backwards (thus mirror imaged words like "WOW" do not make good address labels in UHPS). In our case, this would be ELPMAXE: This actually helps to identify code locations in the symbol tables of the linker RL-51 quite unambiguously.

Finally, it should be noted that a trick has been borrowed from the RATFOR language project[6]. This project was a transition effort to move FORTRAN users over to C. About 90% of the syntax of RATFOR is identical to C. The major differences between the languages is that RATFOR (which is really FORTRAN) must maintain the information passing conventions of FORTRAN, (namely passing by location with the accompanying danger of clobbering variables mentioned in the next section), and the usual

6 Kernighan, Brian W. and Plauger, P. J. (1976)

limitation that FORTRAN does not permit recursive code (i.e. a routine may not call itself).

As part of this approach, for example to insist that constants be given names and reference be made in the program to the name rather than the value so that changes are easilly made in at one location, RATFOR introduces a series of reserved names all in capital letters to symbolically indicate ASCII values. For example, "carriage return" is given the name "CR" and "capital a" is given the name "BIGA", etc. This approach was taken in UHPS in the include file CONSNT.INC where these constant definitions are all collected. The reader who is interested in this is referred to that file.

9.11 The Concept of Structured Programming

Structured programming is a somewhat involved subject yet it is furniture of computer science. It is "the" technique that was used in UHPS to tame the 8051 beast. This section briefly reviews the history leading up to the ideas of structured programming and then defines the structures that evolved from the work of mathematicians in the late 1950's and early 1960's.

9.11a History

Ben Franklin defined man as the toolmaking animal. While this is no longer accepted as uniquely defining the human species, it is nevertheless true that the extent or degree to which man makes tools exceeds probably any other species on the planet. What I mean by this is that man just does not make tools but rather makes tools that make tools that make tools ad infinitum. Since the industrial revolution, the ideal place to see this kind of behavior has been in the machine shop. Now, in the emerging information age, one finds the equivalent in the "software" machine shop where programs are written to aid the development of programs which in turn aid the development of yet more programs. Just as in the machine shop, one would not start with raw materials and expect to directly make something as complicated as a jet aircraft engine, one does not just jump into writing a sophisticated software application without mustering the necessary tools. Some you buy elsewhere and some, because they do not conveniently exist, must be constructed in house.

The first real example of a software tool is the assembler. Modern electronic computers first evolved out of World War II. Originally, they had to be "coded" by hand. This was very tedious and required a lot of repetitious detail - precisely the type of thing that a computer does much better than a human being just as metal filing is done much better by a milling machine. Thus, one day, a frustrated computer "coder" coded his last program by writing a crude assembler that accepted as input a list of mnemonics for the instruction codes and then proceeded to translate these into the appropriate codes along with resolving the address references. Once this was working, the program could be changed to add more features and the current working version used to assemble the new version and thus update itself.

The next logical step came in the mid 1950's when the first so-called "high level" or more human readable languages emerged. In the U.S., these were the business language COBOL and the engineering / science language FORTRAN. This placed the programmer one step further away from the computer and hopefully that much closer to the application being considered. Typically, many such "compilers" generate assembly code as their output which in turn can be assembled and linked to other assembled modules. Several problems emerged in time with these brave software tool techniques. One was that there was the tendency to produce programs that looked more like a huge spaghetti factory out of control. The infamous "GO TO" statement allowed an accidental increase in program complexity that was not needed for completing the task at hand. This complexity of interrelationships lead in turn to increasingly difficult programs to debug and maintain, especially as the demands upon computing increased in time. Programs remained the kind of thing that a "programmer" did and did alone. Rarely did more than one programmer work on the same programming project at the same time[7].

By the 1960's, the concept of modularity and structured programming emerged. Initially this was developed in Germany with Dr. Bauer and his famous team that developed Algol 60. IBM picked up the idea in PL/I and FORTRAN began to adopt it with the WATFOR project. By the end of the decade, the concept of "data hiding" emerged where it was

[7] Cummings, W. C. (1974)

realized that one of the truely important ways to unburden the task of writing large programs was to hide information from different modules on a "need to know" basis much as security agencies do with their employees. This gave rise to the concept of "pass by value" rather than "pass by location" as is done in FORTRAN. FORTRAN is famous for clobbering variables. When variables are passed by location, it is possible for a seemingly buried subroutine to change the value. The result is quite unpredictable and can take forever to debug because the system as a whole has to be debugged rather than single modules.

Passing by value is accomplished by using a stack for passing the values to and from subroutines. It is a bit slower in execution than passing by location and a lot safer. If there is a demonstrated need for speeding execution (which in practice is quite rare), there is nothing to prevent making the variable global and thus directly accessible.

Passing by value eliminates the problem of clobbered variables because in a sense it means that subsequent subroutines have only a sort of "read only" access to the data. Even deeper, in terms of tool making philosophy[8], data hiding facilitates the development of truely general modules which tend to be application independent and as such a library of these evolve in time developed by different programmers so that any given project does not have to start out reinventing the wheel.

By the early 1970's languages such as PASCAL in Europe and C in the U.S. evolved which included all of these features. Just as important was what was not included. PL/I, for example, is a monster language developed by a large committee with everything, probably including the kitchen sink, included. Older languages kept changing to "add" features but in the attempt to satisfy their following, they never changed their underlying philosophy which is the most important advance of all. The newer

[8] Kernighan, Brian W. and Plauger, P. J. (1976)

structured languages were designed from scratch to be tool makers rather than accidental spaghetti factories.

A good example of this is C. C was developed by one man[9]. Basically, it is a universal assembler, allowing convenient contact with the unique registers and hardware of a computer system in the same way that assemblers do and yet its syntax is that of a higher level language. Functional capabilities are done through the use of application libraries that evolve as mentioned above. The main point is that C is written in C and that when a new piece of hardware comes along, one need only hand translate a small runtime package and the well documented code generation tables to accomodate a new instruction set. Thus, suddenly everything that was written in C on an older system will run, just as people expect it to on the new system. The problem of waiting two years for software to emerge after the hardware does[10] is solved and there is no learning curve for the user. These ideas were put dramatically to the test when the original version of the UNIX operating system was translated into C in one summer and installed onto various dissimilar machines throughout Bell Labs[11] thus giving rise to the world's first portable operating system.

9.11b Structures

Mathematicians dominated computer science in the 1960's and out of their way of looking at the subject emerged some interesting ideas. Some of the terminology, such as "vectoring", is now part of the furniture of computer science. Probably one of their greatest influences lies in the development of logical control structures. These structures, when used in a language,

---

[9] Kernighan, Brian W. and Ritchie, Dennis M. (1978)

[10] Cummings, W. C. (1974)

[11] Kernighan, Brian W. and Ritchie, Dennis M. (1978)

eliminate the notorious problem of the "GO TO" statement. What they all have in common is that they have exactly one entry point and exactly one exit point. Thus, debugging is greatly controlled because undesired interconnections within a program are kept to a minimum and when things do go wrong, one debugs by seeking out the offending structure rather than having to keep track of the whole program or major parts of it. The improvement is quite comparable to the invention of assemblers over hand coding.

Mathematicians, being who they are, found that all sequential computer programs could be written using a set of three basic structures. In mathematical terms, the set is complete. Additionally, in practice, one commonly finds three other "nice to have" structures, which while not required for set completeness, nevertheless make programming much easier. Finally, something free comes with all of this, namely that structured programs tend to be "self documenting" in that once one learns how a structure (data or logic control) works and what it means, then reading someone else's code, or even worse, your own code five years later (which I have had to do a lot of recently to write this chapter) becomes possible!

The logic control structures are documented in the following, Figures 9.6 and 9.7, where Sn stands for a statement or sequence of code and Cn stands for a logical condition to be tested while the { }'s indicate the range of a structure.

```
SEQUENCE OR STATEMENT

    {

        S

    }

IF-THEN-ELSE

    if (C) {

        S1

    } else {

        S2

    }

TEST AT THE TOP

    while (C) {

        S

    }
```

Figure 9.6 The three fundamental logic control structures

TEST AT THE BOTTOM

```
repeat {
        S
} until (C)
```

TEST (AND EXIT) IN THE MIDDLE

```
{
        S1
} test (C)
        S2
}
```

CASE:

```
if (C1) {

        S1

} else if (C2) {

        S2
}
        .
        .
        .
 else if (Cn) {

        Sn
} else {

        S(n+1)
}
```

Figure 9.7 The three auxilary logic control structures

Completeness of the set comes from the SEQUENCE structure (which a whole subroutine or even the entire program itself may be considered an example of), the decision structure (IF-THEN-ELSE) and any one of the three loop structures (TEST AT TOP, BOTTOM, or MIDDLE). The other two loop structures are convenient in practice. Similarly, the case structure is a composite of IF-THEN-ELSE structures and is not esscential. It is best implemented with a final default statement (S(n+1)) so that it is clear that if none of the cases tested are true, a known, controlled statement is executed.

9.12 Implementing Program Control Structures in UHPS - the Logic Control Structure Macros (LCSTRC.INC)

As should be evident to the reader, the SEQUENCE structure comes free, all programs are a sequence in their own right, but the others have to be actively developed. In basic UHPS, the decision structure and two of the loop structures are implemented. Again, extensive use was made of the MPL macro facility of MCS-51.

The decision structure in UHPS is made up of two macros, MIF and MELSE. It has the following form:

```
%MIF(VALUE1, OP, VALUE2, LABEL1)

        S1

%MELSE(LABEL1, LABEL2)

        S2

LABEL2:
```

The TEST AT THE TOP structure in UHPS is made up of two macros, MWHILE and MWEND. It has the following form:

```
%MWHILE(VALUE1, OP, VALUE2, LABEL1, LABEL2)

        S

%MWEND(LABEL1, LABEL2)
```

Finally, the TEST AT THE BOTTOM structure in UHPS is made up of two macros, MREPEAT and MUNTIL. It has the following form:

```
%MREPEAT(LABEL1)

      S

%MUNTIL(VALUE1, OP, VALUE2, LABEL1, LABEL2)
```

In all cases, the terms VALUE1 and VALUE2 stand for the two values, either constant or variable, that are being logically compared by the logical operation OP. LABEL1 and LABEL2 are local program location labels that must be unique within a module and are used to set up the scope or range of the logic control structures. It should be noted that this approach permits essentially infinite nesting of structures.

It is instructive to consider how one set of these macros is implemented. Figure 9.8 documents the implementation of the decision structure:

```
; if (CONDITION) {  }

%*DEFINE (MIF(X,CONDITION,Y,ELSELAB)) LOCAL LABEL
(
        %%CONDITION(%X,%Y)
        JNC  %ELESELAB
)

; else {  }

%*DEFINE(MELSE(ELSELAB,ENDLAB)) LOCAL LABEL
(
        JMP  %ENDLAB
        %ELSELAB:
)
```

Figure 9.8 The decision structure in UHPS

The complete collection of these macros implementing the different control structures makes up the contents of the include file LCSTRC.INC to which the curious reader is referred.

9.13 "External" Memory Macros in UHPS

It has been emphasized before that the 8051 has an extremely small internal data memory. A good part of it is completely reserved for control of the special hardware in the 8051. The remainder is left for one or more register banks, a stack, and any program variables, either local or global. It did not take long before the optical module program exceeded this limit in spite of considerable effort to minimize this possiblity (while retaining the flexibility of the structured language approach mentioned above). The only commands in the 8051 instruction set that deal with the external memory are the ones for moving a byte in and out of external memory and the accumulator. The impact of this was that a whole series of macros, similar to those already developed had to be written so that the use of external memory would be virtually just as automatic in UHPS as the use of internal memory. The desire was that the syntax be totally compatible so that routines that had already been written and used in internal memory could be easily modified for use in external memory. These extensions are found in the include file RUNMAC.INC.

First, it was necessary to develope external data handling macros that are analogous to the 8051 MOV operation. Five of these were brought up as shown in Table 9.7:

Table 9.7

The external data MOV macros

Macro	Description
TMOV	move "to" DATA from XDATA
FMOV	move "from" DATA to XDATA
TAMOV	move "to" <A> from XDATA
FAMOV	move "from" <A> to XDATA
EMOV	move XDATA to XDATA

Again, it is illustrative to see how one of these macros is defined. Figure 9.9 documents one of these:

```
; move an XDATA source byte (X) to a DATA
; destination (D)

%*DEFINE(TMOV(D,X)) LOCAL LABEL
(
        MOV   DPTR,#%X
        MOVX  A,@DPTR
        MOV   %D,A
)
```

Figure 9.9 The TMOV macro definition

With these in place, the extended decision macros are defined. For each of the six decision primatives, each with its corresponding (internal data) decision macro, three analogous extended decision macros had to be defined. This was so that the three cases of comparing constant with variable, variable with variable, and variable with constant could be implemented without the need to first force the constant by hand into a reserved register (either R0 or R1 as the case may be) before invoking the macro. For example, the analogous macros for MXGTY are (note that "K" stands for constant and "E" stands for external):

Table 9.8

The XGTY extended macros

Macro	Description
MXGTY	internal memory macro
EXGTY	external variable to variable
EXGTK	external variable to constant
EKGTY	external constant to variable

To illustrate these macros, Figure 9.10 documents the EXGTK macro definition:

```
%*DEFINE(EXGTK(X,K)) LOCAL LABEL
(
        %TMOV(R0,%X)
        MOV  R1,%K
        CALL XGTY
)
```

Figure 9.10 The EXGTK macro definition

Along this same line, namely that we are working in the case of the external data memory with an instruction set that is even crude compared to the low standards of the 8051, it made sense to develope additional macros that would make up for some of the most important missing operations. The operations considered were the stack handling PUSH and POP, the ability to increment or decrement an external data value, and the ability to perform basic arithmetic and logic operations on an external data value with what has already been moved into the accumulator. In other works, these are extensions of internal data operations that are not even as powerful as the minimal ABX model. Table 9.9 documents these macros:

Table 9.9

Minimal extension macros for handling external memory data

Macro	Description
EPUSH	push value on XDATA stack
EPOP	pop value off XDATA stack
EINC	increment an XDATA value
EDEC	decrement an XDATA value
EADD	add XDATA value to <A>
EADDC	add XDATA value to <A> + <C>
ESUBB	subtract XDATA value from <A> - <C>
EAND	"and" XDATA value with <A>
EOR	"or" XDATA value with <A>

Again, the remaining details of implementation are all to be found in the file RUNMAC.INC. It should be obvious that this process could be continued to develop a system on the 8051 that mostly used the much larger external memory. This system could be modeled after the ABX mimimal model. The internal data memory could be reserved for the frequently called primatives while the application would entirely exist in the external memory. Combining these ideas with structured programming would make a more reasonable programming environment than the current implementation of UHPS which is still more bound to the internal data memory than the external data memory.

9.14 The (Library) Primatives of UHPS

The subject of primatives has been mentioned before but not really explained. In the context of UHPS, these are a collection of special subroutines which form a sort of core to doing anything practical on the 8051. Structured programming hides data and this is done by passing by values through the use of a stack. Since the primatives are omnipresent for all applications and since many of them are directly dependent upon one another, they do not make use of the stack. They were very carefully debugged independent of any application.

Instead of the stack, each group of primatives, and there are four distinct groups of primatives, has its own programming model by which they interact with one another. Typically, they make use of the eight registers in the register bank at the very beginning of the internal data memory. Already, we have seen this in the case of the six decision primatives found in the file RUNLIB.P51. It should be noted here, that the file extension P51 was used to indicate primatives whereas A51 was used to indicate a single 8051 subroutine and in a few rare cases, S51 was used to indicate multiple 8051 subroutines in the single file (this was done for some "middle level" subroutines that worked intimately with one another yet did not warrant being made primatives nor were they specific enough to be considered part of the application program). These collections of primatives in a single file are called libraries in UHPS and are listed in Table 9.10:

Table 9.10

The libraries (of primatives) in UHPS

Library file name	Number of routines	Integrated function
COMLIB.P51	11	serial communications
RUNTIM.P51	6	decision
TIMER.P51	5	software timing
TABLE.P51	3	table handling

The programming model for the decision primatives has already been explained. What will be considered here for the rest of this section is the programming models for the other three libraries and a very brief description of the primatives found in those libraries.

The serial communications library COMLIB.P51 consists of 11 primatives. These primatives are listed in Table 9.11 (note that <CR> means the ASCII carriage return and <LF> means the ASCII line feed characters):

Table 9.11

The serial communications library primatives

Primative	Function
PUT_CHAR	output a character
OUT_CHAR	PUT_CHAR with a data strobe
GET_CHAR	input a character
IN_CHAR	GET_CHAR with time out
PUT_CRLF	output <CR> + <LF> characters
OUT_CRLF	version of PUT_CHAR using OUT_CHAR
PUT_STRING	outputs null terminated (CODE) string
OUT_STRING	version of PUT_STRING using OUT_CHAR
PUT_DATA_STR	outputs a (DATA) string of known length
GET_NUM	inputs a four character numeric string
ECHO	inputs and outputs a character

Most of these primatives repeatedly call upon the fundamental primatives GET_CHAR and PUT_CHAR (or their variations). The programming model for the these primatives is documented in Figure 9.10:

Register	Usage or contents
<A>	input/output character
<R0>	data string address
<R1>	length of data string
<DPTR>	code string address

Figure 9.11 The programming model for the serial communications primatives

The software timer library TIMER.P51 consists of five primatives. These primatives are listed in Table 9.12:

Table 9.12

The software timer library primatives

Primative	Function
DELAY	busy wait 4 microseconds
MDELAY	busy wait 1 millisecond
SDELAY	busy wait 1 second
MINDEL	busy wait 1 minute
HDELAY	busy wait 1 hour

The programming model for the these primatives is actually an extension of the one for the serial communications primatives. The extensions are documented in Figure 9.12:

Register	Usage or contents
<R2>	microsecond clock
<R3>	microsecond counter
<R4>	millisecond clock
<R5>	millisecond counter
<R6>	second clock
<R7>	second counter

Figure 9.12 The programming model for the software timer primatives

It should be noted that this is not the most efficient use of the registers (see comment in the file TIMER.P51) but it was adequate at the time of development for the purposes intended.

The external data table handling library TABLE.P51 consists of three primatives. These primatives control access to a table no longer than 256 bytes long in the external data memory and are listed in Table 9.13:

Table 9.13

The external data handling library primatives

Primative	Function
POINT	point to a table entry
STORE	store the value in <A> at the point in the table
FETCH	fetch the value at the point in the able and transfer it to <A>

The programming model for the these primatives is documented in Figure 9.13:

Register	Usage or contents
<R0>	lower address of byte
<R1>	upper address of byte
<R2>	offset from beginning of table

Figure 9.13 The programming model for the table handling primatives

9.15 The Subroutines of UHPS

So far, the conventions, macros, and primatives of UHPS have been discussed in detail. This leaves one last remaining topic, the intrinsic subroutines of UHPS. Unlike the primatives, these routines make use of the system stack for parameter passing. Basically, there are two ways by which a subroutine can work its way directly into the language rather than remain in an application library. One is that it does something intrinsic with the extended hardware of just about every application that was envisioned and the other is that its function is so common that it deserves to be made part of the language. Admittedly these criteria are arbitrary and all of this could be left in an application library. Basicly, they were not because of the desire in bringing up an application to limit the total length of code in the modules being handled on the single disk drive MCS-225A system to as few as necessary at any given time. These are summarized in the following table:

Table 9.14

The intinsic subroutines of UHPS

Subroutine name	File name	Description of function
INITIAL	INITAL.A51	initialize 8051 stack and 300 baud communications
ISALNO	ISALNO.A51	check if character is alphanumeric
ISHEX	ISHEX.A51	check if character is hexidecimal
ASCBIN	ASCBIN.A51	convert ASCII character to binary
HEXBIN	HEXBIN.S51	convert hexidecimal character to binary
HEXNIB	HEXBIN.S51	assists HEXBIN
BINHEX	BINHEX.S51	convert binary to hexidecimal character
NIBHEX	BINHEX.S51	assists BINHEX

All of these routines were originally brought up within the internal data memory. Several of them (HEXBIN, HEXNIB, BINHEX, NIBHEX, and ASCBIN) have been translated into the external data memory and the most of the rest ought to be.

This completes the documentation of UHPS. The full code is found in Appendix 6. It should be noted that UHPS (pronounced "ooops" as in "ooops, we need a language") was written as a sort of desparation move. It evolved as understanding of the many problems associated with programming the 8051 family microcontroller evolved. There is much room for improvement yet it did the job and is a good illustration of the application of a number of software techniques that can be used to convert a crude development system into something considerably more precise somewhat equivalent of going from that hand file to the milling machine.

CHAPTER 10 – Instrument Building Part V – The Design of the SBC Central Controller Circuit

10.1 Introduction

This chapter is the fifth and last of the series on instrument building of the Short Prototype String. The main concern of this chapter is to document the hardware and software of the single board microcomputer based central controller circuit for the String Bottom Controller. This circuit is intrinsically part of the SBC circuitry which is housed inside the large cylindrical pressure housing.

By the Fall of 1985, the DUMAND project had progressed to the point where the basic SPS was ready for its first attempt at system integration in the form of a cruise. The optical modules existed, the basic power transfer system existed, the single mode optical fiber system existed, the ship board command and control computer system existed and so it was decided to tie these all together. A number of critical items did not exist, most importantly the high speed electronics of the SBC. Nevertheless, a crude "Ersatz" SBC for the purpose of doing simple minded coincidence work was developed. Along with this, a command and control computer for the Ersatz SBC was developed to control this.

As before, the INTEL 8051 family microcontroller was used simply because a development system now existed in house. The microcontroller board was brought up from scratch and again programmed in UHPS. It was capable of intercepting messages intended for it and replying in a definitive manner. Functionally, it controlled the latches on the Ersatz SBC coincidence logic board and the interface to the sensor logic board. The latter, consisted of an eight bit wide, eight channel, A to D sensor board designed somewhat similar to what was used in the optical modules. This acted as the interface to a series of sensors for such things as temperature and voltage levels within the Ersatz SBC electronics.

That "cruise" was pretty futile. However, in the attempt at systems integration, much was learned. Among the lessons was the realization that controlling the full set of electronics involved in the SBC would easily overtax the abilities of the 8051 microcontroller. Additionally, there would be a real need to develop a lot of support hardware for all of the areas, communications interface, system sensing, and latch control would be considerably more involved in the real SBC than they were in the Ersatz SBC exercise. One advantage however was that the space available inside the instrument housing was not as limited as in the case of the optical modules. With all of this in mind, the search was begun to identify an off the shelf computer that could serve as the central brain for the SBC that was a lot easier to program. Once found, then all of the effort could be spent in designing and testing the required and expanded system interfaces to it and in writing its executive control program rather than having to design an adequate computer from scratch or to worry about having to write and maintain the basic programming language.

10.2 The SBC Microprocessor System Overview

As was mentioned, a number of lessons were learned from the system integration exercise of the Ersatz cruise of the Fall of 1985. Several of the lessons need to be briefly pointed out since they formed the motivational core for some of the design of the SBC computer control system.

First, it was found that the assumptions previously made in how the fiber optics cable would behave for signal passing purposes were totally wrong. It had been assumed that the cable was sufficiently like a coax cable that a commercial modem system used in the earlier MUON String experiment that uses a one megahertz carrier would handle the requirement. Actual tests on the cable showed that frequency transmission was relatively linear until around 8 to 10 KHz where it rapidly rolled off to nothing. Thus, the carrier, let alone the signal, would never get through the cable. A panic drive to produce an in house modem system increased attention to communications throughput in the SPS.

Another lesson was the power supplies in the SBC housing. These had to supply a lot of current (the fast electronics of the SBC are all ECL) in a small space. Thus, the original linear supplies had to be changed to switchers. The problem is that the way power is first applied to the SBC and how it continues to be supplied when changes are made in the current demand require careful consideration. Applied voltage is ramped up slowly from on board ship since the actual voltage to be found in the SBC will vary depending upon the load. With the switching supplies one has to have a procedure for monitoring this.

Switching supplies are tricky. They must have a supply voltage that is within a reasonably narrow range in order for them to start switching. Thus, the approach taken was one of setting the ship board supply to an approximate level, then snap on the power, and then hope to be able to monitor the power from the SBC. Monitoring required that the system started up within the acceptable range of the switchers so that they in turn output voltages which allowed the SBC electronics to come on, particularly the controlling computer. With the controlling computer then monitoring the actual AC power supplied, the ship board controls could be adjusted to nominal values. Every time a major change in the power requirements was made, the ship board controls had to be adjusted again.

Concerning communications in the Ersatz cruise, the 8051 microcontroller controlling the Ersatz SBC sat in parallel just like the optical modules on the string. Similarly, the power distribution microcontroller was connected in parallel. The main reason for this is that with the 8051 family of microcontrollers, one had no other choice. The 8051 has only the equivalent of a single duplex serial communications channel.

In this integration exercise, it was found that the environmental module (built elsewhere) had no provisions for making any replies to order wire commands. Additionally, there was no provision made to handle the great flood problem. Thus, since the module never "talked", there was no screamer problem but at the same time there was no reply information and a real possibility of losing the entire communications channel due to an implosion of the environmental module controller sphere. The connections to the string were modified to include potted resistors to prevent the great flood problem but getting the module to reply to commands turned out to be impossible because among other things the source code to the ROM, according to the author, was "no longer available". Thus, ultimately a requirement of the SBC control computer was to fake replies from the environmental module in order to satisfy the ship board computer and to at least verify that the command got down through the system at least as far as the SBC.

Figure 10.1 symbolically shows the interface specification for the SBC control computer within the SPS as a whole and the rest of the SBC in particular.

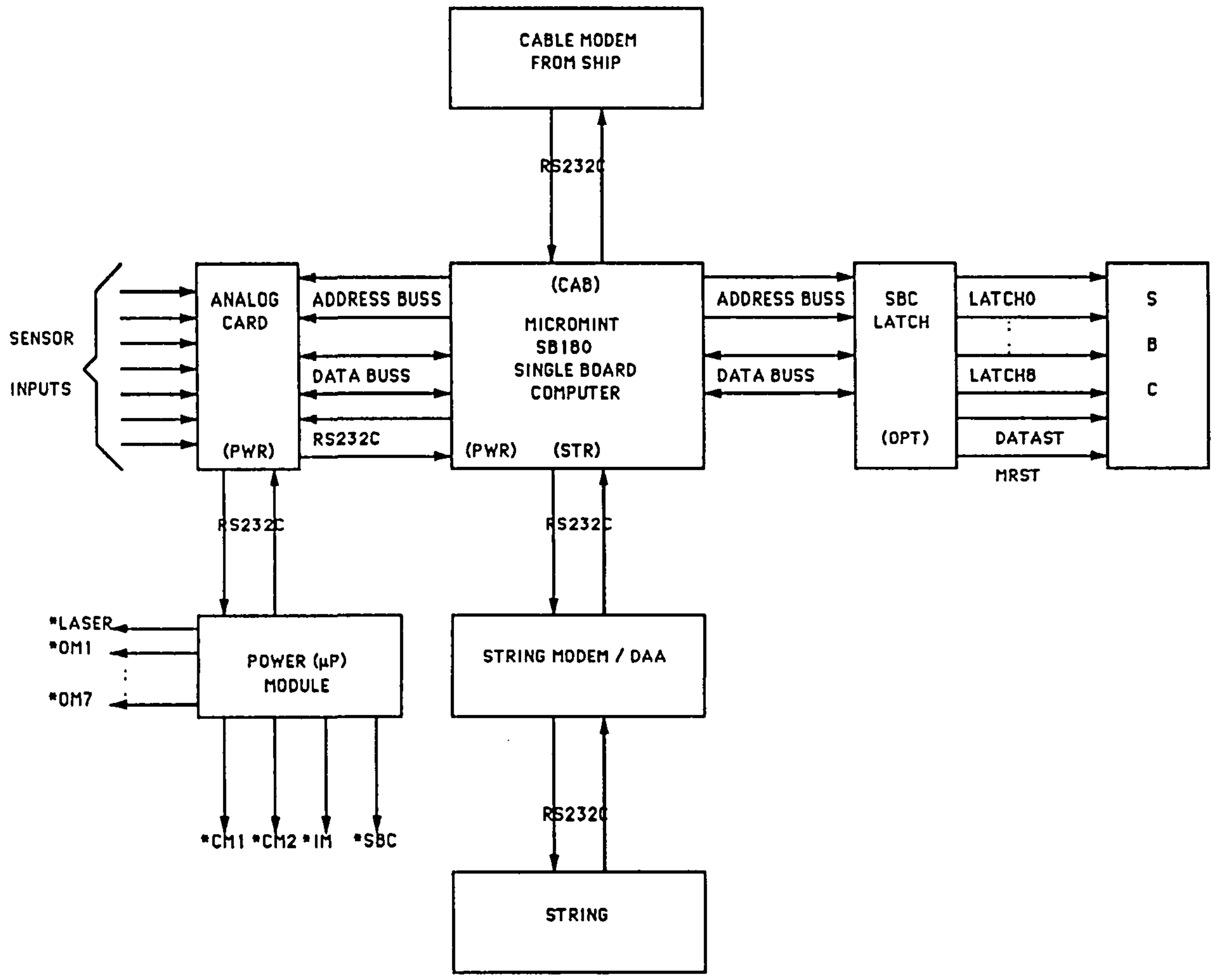


Figure 10.1 The interface specification for the central control computer in the SPS

As one can see, there are five specific interfaces required: (1) the string modem, (2) the ship's cable modem, (3) the SBC high speed electronics, (4) the analog sensors, and (5) the power control circuit. Each of these in turn will be documented. However, it makes sense to document the control computer itself first.

Before proceeding with that, it should be pointed out, that a major decision was made as a result of the Ersatz cruise that the SBC control computer would not just sit on the string in parallel with the other modules but instead would sit in series between the fiber optic cable from the ship and the string. Since this computer directly controls the latches to the SBC fast electronics, if it were to ever go down the whole system would go down. As such, placing the computer in series was not considered to be a real change in system reliability. The advantage was that all those little annoying

10.3 The Hitachi HD64180 Microcomputer

A "single board" computer, only 4 by 7 1/2 inches in size was identified. This computer, designated the SB180 by its manufacturer, Micromint Corporation, required very little in the way of power, came loaded with hardware and software features, and best of all easily performed double duty by being its own development system. The software included a very sophisticated ROM monitor (which was modified to jump into the SBC control program upon receiving a start up command), a truly advanced operating system, about 100 system utilities including an assembler and linker, and was easily programmed using a commercially available compiler. In this exercise, unlike before, the wheel did not have to be reinvented!

Central to the SB180 computer board is its microcomputer chip, the Hitachi HD64180. The SB180 was originally announced in September 1985, a few months after the chip became available in any quantity. It is probably even to date the most sophisticated eight bit microcomputer chip available anywhere in terms of the amount of built in hardware. Immediate comparisons to the INTEL 8051 are striking. For example, the system clock is a 6.144 MHz derived from a 12.288 MHz crystal. Thus, the HD64180 runs a good six times faster than the 8051 and with its "real" Z80 instruction set, has computing power that is probably an order of magnitude better. As such, to understand the SB180 it is really best to first look at the HD64180 in some detail. Figure 10.2 is a block diagram of this chip.

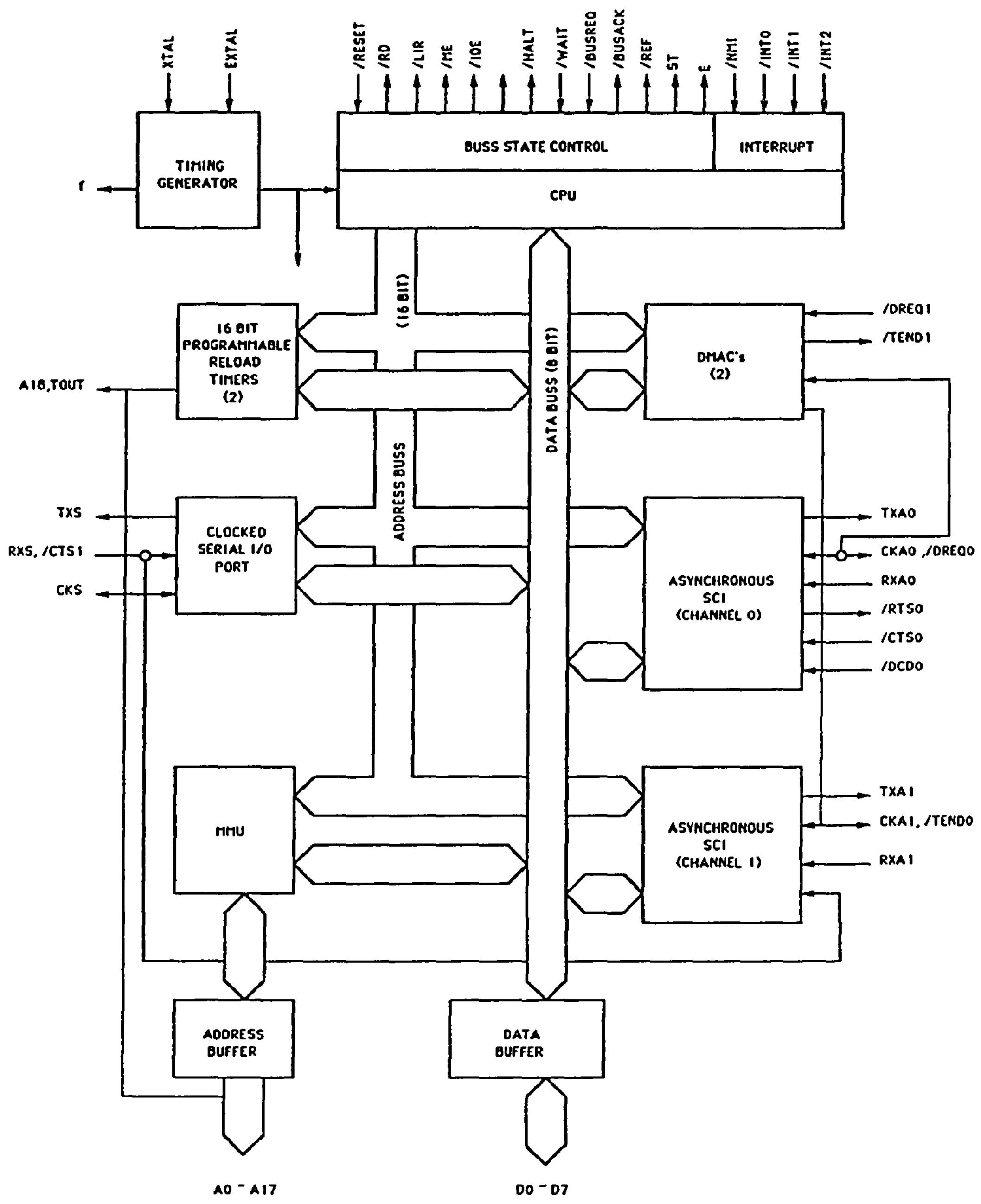


Figure 10.2 Block diagram of the Hitachi HD64180

The Hitachi HD64180 is implemented in CMOS technology which means low-power operation (battery operation is possible). The instruction set of the HD64180 is an enhanced version of the famous Zilog Z80 thus immediately coupling it to a large existent software base. The Z80, like most of the rest of the computer universe, is based upon the Von Neumann architecture combining the instruction space and data space into a common memory space. Thus, the problems of the Harvard architecture mentioned before with the INTEL 8051 family designs are not a problem here. Unlike the Z80 which has only 16 address lines limiting its memory space to 2^{16} or 64 kilobytes of memory, the HD64180 has 19 address lines increasing the addressable memory size to 512 kilobytes. A built in memory management unit (MMU) helps the system programmer to access this memory in a variety of dynamic ways overcoming some of the limitations of the Z80 instruction set which uses (as does the 8051) absolute rather than relative code. In other words, with the MMU, code can actually reside in one area of memory where it could not be executed because of its specific address references yet the MMU will patch this up so that the central processor unit (CPU) can run this anyway. Figure 10.2 shows that the HD64180 is made up of nine functional blocks. These blocks fall into two broad categories, those associated with processing and those associated with the integrated I/O resources of the chip.

10.3a The Processing Function Blocks

The first processing block is the central processing unit (CPU). As mentioned, it contains the microcode for an extended version of the Z80 instruction set. The added 12 instructions are only those necessary for the direct control of the on-chip hardware that makes the HD64180 such an outstanding microcomputer. Many of the Z80 instructions actually require fewer clock cycles than on a standard Z80 so that enhanced execution results even at the same clock rate.

The second processing block is the timing generator. The internal system clock is generated here from an external source such as either a crystal or clock input. Slaved to this is some prescaling logic which derives other clocks necessary for the support of on-chip I/O and system support devices such as asynchronous communications. As such, since timing standards are involved, the driving frequency must be some magic number which would then result in standard clock frequencies for communications. Standard data transmission rates are derivable from internal clock rates (always half the rate of the external source) of 3.072, 4.608, 6.144 (the usual one), and 9.216 MHz.

The third processing block is the bus-state controller. This block performs all status and control functions for bus activity. This includes the external bus-cycle wait-state timing, $\overline{\text{RESET}}$, refresh of any dynamic random access memory, and master direct memory exchange. There are two defacto standards for the control of peripheral devices, namely those of Motorola and INTEL. Rather than take sides in an endless debate over which approach is better, this block generates the control signals for both.

The fourth processing block is the interrupt controller. It is capable of monitoring and assigning interrupts to up to four external and eight internal devices. There exist a variety of programmable interrupt response modes.

The fifth and last processing block is the memory management unit. As indicated before, this device performs the miracle of coupling an instruction set that only knows about a 64 kilobyte (logical) memory space to a 512 kilobyte physical memory space.

10.3b The Integrated I/O Functional Blocks

The first integrated I/O block is the direct memory access controller (DMAC). The function of direct memory access is often involved yet it is very useful. In some way, the goal is to trick the CPU into allowing another device direct access to the memory. The reason this can be advantageous is that in the case of frequent or large memory transfers, this can be faster than passing everything through the CPU. There are two classical approaches.

This first one, cycle stealing, is used to handle the problem of short but frequent memory transfers. Basically, the idea is that the CPU does not get all of the cycles from the clock. There is a bit of logic which can, upon receiving the proper request, redirect some memory cycles to some other piece of hardware which then either reads or writes directly from or to system memory.

The second one, burst DMA, is used to handle the problem of large memory transfers. Here, instead of the CPU skipping a few cycles while it executes on never knowing that the memory has been played with, the CPU is actually halted. Then, control of the memory is given over to the other device which needs access.

The DMAC of the HD64180 handles both kinds of DMA. It is a two channel device capable of memory-to-memory, memory-to-I/O, and memory-to-memory-mapped-I/O transfer. It is optionally edge or level sensitive on the send-request input handling address increment, address decrement, and address no change type transfers. It directly accesses the full 512 kilobytes of physical memory and transfers up to 64 kilobytes at a time in burst mode. Such transfers are accomplished at a rate of one megabyte per second.

The second integrated I/O block is the asynchronous serial communications interface (ASCI). The ASCI has two universal asynchronous receiver transmitter channels for serial communications. These are fully duplex so that the communications on both channels is bidirectional (compare this to the 8051 which has two unidirectional channels so that only one bidirectional channel is possible). Additionally, the ASCI includes a programmable data transmission rate generator, modem control signals, and a multiprocessor communication format. Further, the ASCI can hook directly into the DMAC for high speed serial data transfer unburdening the CPU.

The third integrated I/O block is the clocked serial I/O port (CSIO). The CSIO provides a single synchronous simplex (half-duplex) serial transmitter and receiver. This is most useful as a channel for high speed communications with another computer.

The fourth and final integrated I/O block is the programmable reload timer (PRT). Functionally it provides two programmable 16 bit timer-counters each consisting of an I/O line, a 16 bit timer data register, and a 16 bit timer reload register. The base clock for these timers is the system clock divided by 20. One of the PRT channels can be optionally programmed as a waveform generator. Unlike the 8051, it should be noted that the timer-counters are independent of the serial I/O function so it is no longer the case of having to choose which function you want, both are always available.

10.4 SB180 Single Board Computer Description

The SB180 computer was designed to be just about the most versatile computer possible at the time of its design (1985). To begin with, its computer engine is the Hitachi HD64180 which, as has already been shown, has an enormous amount of built in hardware yet couples directly into an already existent software base in the form the Z80 instruction set. Aside from the power supply, the rest of the SB180 is the collection of interfaces between the HD64180 and the peripheral functions of the board. These functions include two RS-232C serial I/O interfaces, memory interface, dynamic memory refresh, centronics (parallel) printer interface, floppy disk interface, and an expansion bus. Figure 10.3 is a block diagram of the functions found in the SB180 computer.

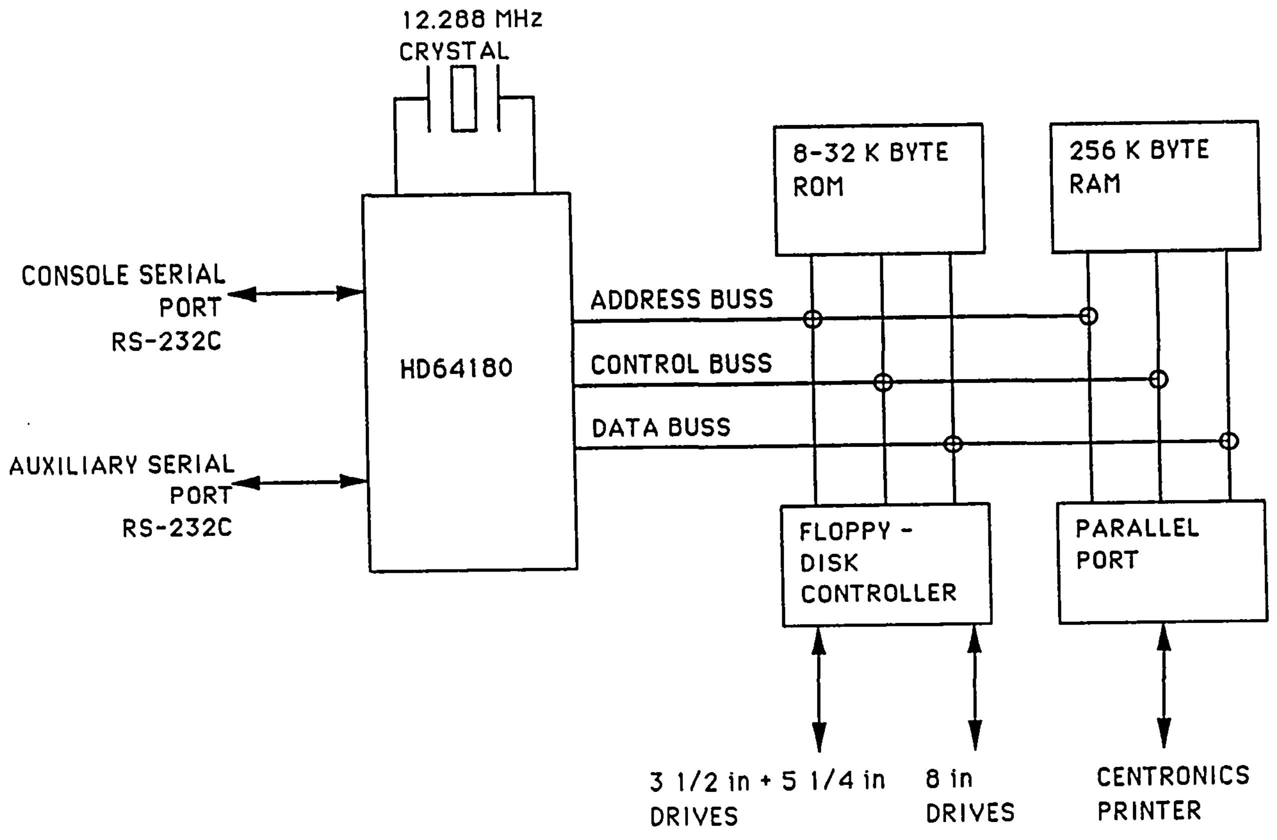


Figure 10.3 Block diagram of the SB180 computer

The RS-232C serial I/O interface consists of two full duplex serial I/O channels operating at RS-232 logic levels. The circuit is simply an extension of the ASCI ports of the HD64180 directed through level shifters from TTL to RS-232. Channel 1 is used for the system console and channel 0 is used as an auxiliary port for serial printers, telecommunications modems, etc. This channel has some additional handshake signals available with it so that it can more readily interface with such devices if necessary. What is very nice, is that with the two serial communications ports, the SB180 can be used as its own development system since a spare port is available for debugging communications problems while the other port is used for console control. These ports are all completely programmable so that they can easily be adapted in terms of baud rate, parity, number of bits in a character, etc. for a given operation.

The memory interface includes a 28 pin boot ROM socket which, depending upon the settings of a series of jumpers, can accommodate eight bit wide by eight kilobyte, by 16 kilobyte, or by 32 kilobyte memory devices. The boot ROM which comes with the system is very sophisticated and includes the disk boot code and a ROM monitor for stand alone operation. This ROM occupies the lowest (256 kilobyte half) part of memory. Upon RESET, the HD64180 begins execution at physical address 00000 (hex), the start of the boot ROM.

Table 10.1

The boot ROM monitor commands.

A	ASCII table: Prints an ASCII table
B	Bank select: Selects a 64k-byte memory bank
C	Copy disk: Systems with 256k bytes of RAM can perform single drive copies
D	Display memory: Displays memory in hexadecimal and ASCII
E	Emulate terminal: Console keyboard is echoed to the auxiliary RS-232C output and the RS-232C input is echoed on the console display
F	Fill memory: ANy portion of memory is filled with a data byte
G	Go to program: Starts program execution at specified address and optionally includes a breakpoint
H	Hexmath: Prints the 20 bit sum and difference and 32 bit product of the two arguments
I	Input port: Prints the 8 bit data input from specified port
K	Klean disk: Formats a specified drive
M	Move memory: Moves a block of memory
N	New command: Enables new commands from extended ROM space
O	Output port: Byte is output to specified port address
P	Printer select: Toggles printer selection between the Centronics parallel port and the auxiliary RS-232C port
Q	Query memory: Searches memory for pattern of 1 to 4 bytes
R	Read disk: Reads specified sectors form drive into memory
S	Set memory: Displays memory contents and allows new data to be entered
T	Test system: Tests various system devices

Table 10.1 (Continued)

The boot ROM monitor commands.

U	Upload hexadecimal file: Uploads Intel hexidecimal file from auxiliary or console serial port
V	Verify memory: Compares two blocks of memory
W	Write disk: Writes specified sectors to disk from memory
X	eXamine CPU registers: Displays main and alternate CPU registers and prompts for modification of main registers
Y	Yank I/O registers: Displays the HD64180 on-chip I/O registers
Z	Z-System boot: Boots the Z-System form disk

The dynamic RAM is located in the upper 256 kilobyte half of the physical memory space. It requires refreshing at a rate of 256 refresh cycles (eight bits so $2^8 = 256$) every 4 milliseconds. The refresh signals are directly produced by the HD64180. Appropriate control strobes for refresh, memory read, and memory write are all generated by the HD64180 and brought out to the RAM along with the required 18 address lines and eight data lines. It should be noted here, that unlike INTEL microprocessors and microcontrollers like the 8051, the data and lower address lines are not multiplexed so there is no need for a demultiplexing circuit. As designed, the refresh overhead in the SB180 is only 2.5% or two out of 80 cycles.

The Centronics parallel printer interface is consists of an eight bit latch and a flip-flop located at 0C0H through 0C1H. Printing is accomplished by first writing data to the port at 0C1H. This sets up the data in the latch for the printer and then asserts the printer strobe signal. Writing data to the port at 0C0H deasserts the printer strobe signal. When the printer has finished grabbing the data, it handshakes by returning an acknowledgement strobe. In turn, this strobe, when received by the SB180 is redirected to the CPU as an interrupt signal. The interrupts handler clears the interrupt by writing a dummy output to the port at 0C0. Since this interface is interrupt driven, operations such as background spooling become relatively easy to implement.

The floppy disk interface is very versatile. Due to the choice of the floppy disk controller chip. The one used is the SMC 9266. It includes such functions as data separation and programmable write precompensation. The net result of this interface is that it can directly control any softsectored floppy disk drive single or double sided; single, double, or quad density; 3 1/4, 5 1/2, or 8 in diameter. Up to four disk drives are supported simultaneously.

Specifically, two interface busses for floppy disk drives are provided. The 50 pin buss interfaces to 8 disk drives. The 34 pin buss interfaces to either the 5 1/2 or 3 1/4 disk drives. The accompanying software allows a choice of a number of different soft sectored floppy disk formating conventions to be used.

The external buss (XBUS) interface consists simply of all of the major address, data, and control signals being routed to one 32 pin connector along with a bus valid strobe line. As mentioned before, the HD64180 provides control signals for both Motorola and INTEL type peripherals and these are found here.

Finally, some mention should be made of the power supply interface. The SB180 requires only about 1 amp at +5 VDC and 10 milliamps or so at +12 VDC. The +12 VDC is only used for the RS-232C interface which also requires a negative logic voltage. This is generated on board through the combination of a +9 VDC Zener diode and a small inverter.

10.5 Description of the SBC String Modem Card

The string modem card is the communications interface between the SB180 computer board and the string of optical, calibration, and environmental modules. All commands sent down from the ship board computer are first intercepted by the SB180. In turn, those commands that are not intended directly for the SB180 are relayed to the string and the power distribution module which logically all sit in a parallel communications mode awaiting a command for one or more of the modules to execute. These commands are all transfered at the order wire communications rate of 300 baud.

As shown in Figure 10.1, the interface between the string modem card and the SB180 computer is an RS-232C connection. This greatly simplified debugging and maintenance since ASCII terminals could be hooked into the system to monitor activity directly. The auxilary (AUX) channel of the SB180 (named in the software STR for the string) was dedicated to this task.

The string modem board was mounted parallel to the end caps of the SBC housing, i.e. perpendicular to length dimension of the cylinder. The actual board that the string modem circuitry was built on was a modified S100 prototype edge connector board shaped to roughly fit the circular housing of the SBC with a 22/44 pin interface with 1/8 inch spacing. The string modem circuit was similar to the modem circuits found in the optical and calibration modules. It consisted of two parts, a modem circuit that went from an RS-232C serial communications system to a 300 baud FSK communications system. The FSK system selected was the mirror image of all of the ones found on the string. In other words, where all of the string modems were set to the "answer" mode, the SBC string modem was set to the "originate" mode[1].

[1] See Table 6.1

Additionally, the data carrier detect (DCD) signal was brought out of the modem chip through the RS-232C interconnection to the SB180 to facilitate handshaking. Unlike the optical modules which can all listen to a common carrier and signal from the single source of the SBC, the SBC modem has a potential ten different sources to listen to. If more than one of these transmit a carrier, the receive section of the SBC string modem becomes confused since there is no guarantee that the different carrier sources are going to be in phase. The result is a disaster for communications. As such, the individual string modules only transmit a carrier at the time they are going to transmit a reply. The DCD output of the modem is a control signal letting the SB180 know that what is coming from the modem during the time that DCD is active is real communications and not just random noise.

The other circuit is the interface between the modem and the string itself. This is the Data Access Arrangement (DAA), again, quite similar to that found in the optical and calibration modules. Figure 10.4 documents all of this.

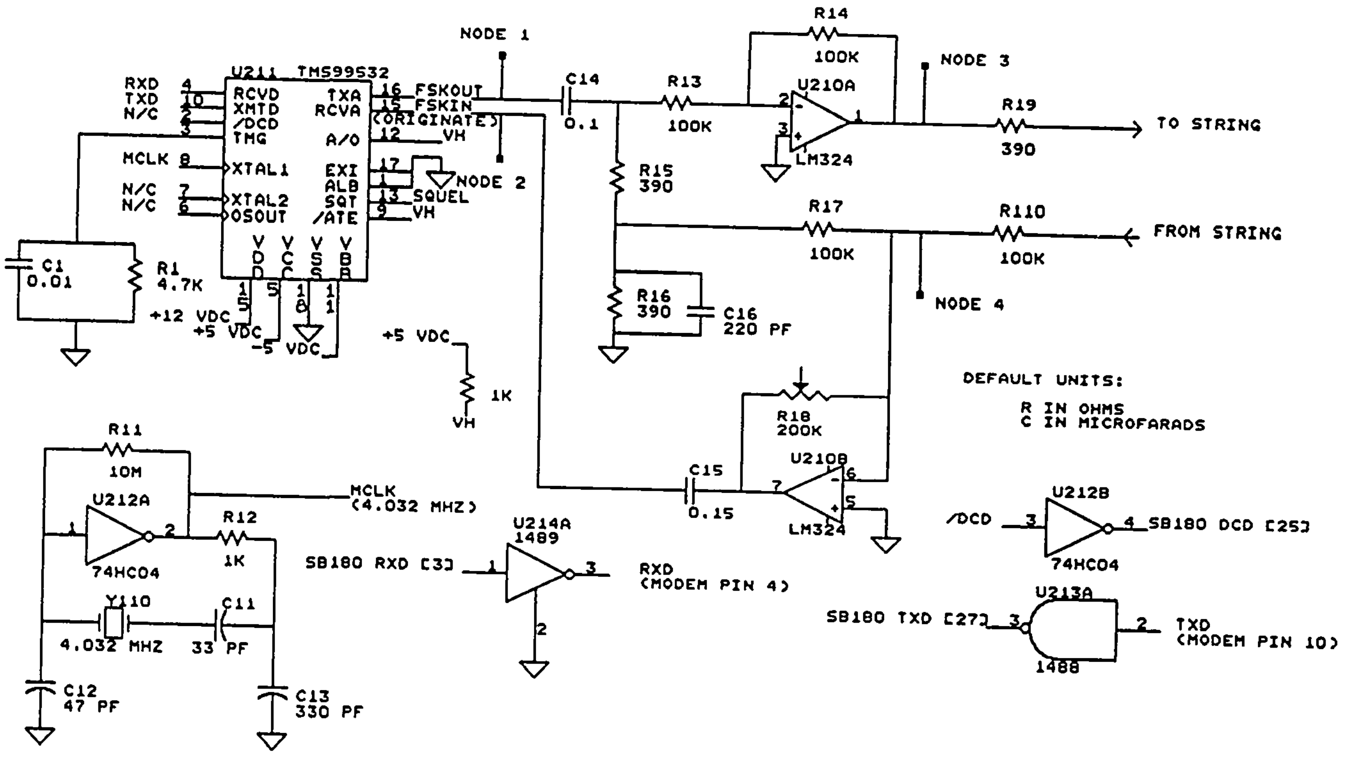


Figure 10.4 The SBC string modem with three wire based DAA interface

The DAA circuit shown in Figure 10.4 consists of two LM324 operational amplifiers arranged in a sort of a bridge network so that the receiver (node 4) does not see the transmitter (node 1) during transmission. By itself, the first amplifier acts as a transmission amplifier arranged as a unity gain buffer driving a 390 Ω load (as the string at node 3 will appear to it). There are two pathways to the receiver. The first is from the string through a unity gain buffer identical to that used in the transmitter. The second comes straight from the transmitter through 100 Ω and 100 K series resistors exactly matching the current (due to local transmission) directly from the output of the transmitter amplifier to the input of the receiver amplifier. Since the transmitter amplifier is an "inverter", the phase of these two transmission signals theoretically should exactly cancel at the input to the receiver amplifier. Of course, the phase inversion going through all of the components is not exactly 180 degrees out but it is pretty close, especially if the unity gain buffer of the transmission amplifier is properly adjusted so the feedback signal can be smaller than any expected receiver signal from the string and properly adjusting the receiver amplifier gain can separate the two. Since the SBC modem is the origin of all commands sent to the string, its transmit line is the common receive line of the string modules. Likewise, the receive line of the SBC modem is the common transmit line of the string modules.

The DCD scheme is dependent upon associated turn-on and turn-off times. Internal to the TMS99532 modem chip is an Automatic Gain Control (AGC) function which, among other things, compares the level of the received signal to the threshold of the AGC[2]. Should the signal be above the threshold, then carrier has been detected. However, occasional errors due to poor signal quality do not warrant disconnecting the circuit. To prevent this, a time out function is supplied which will keep the DCD line its current level until the time out is over. This way, transitions between the carrier detect and no carrier detect conditions are debounced in time. Two external components, a resistor R20 and a capacitor C20, wired in parallel

[2] TMS99532 Application Report

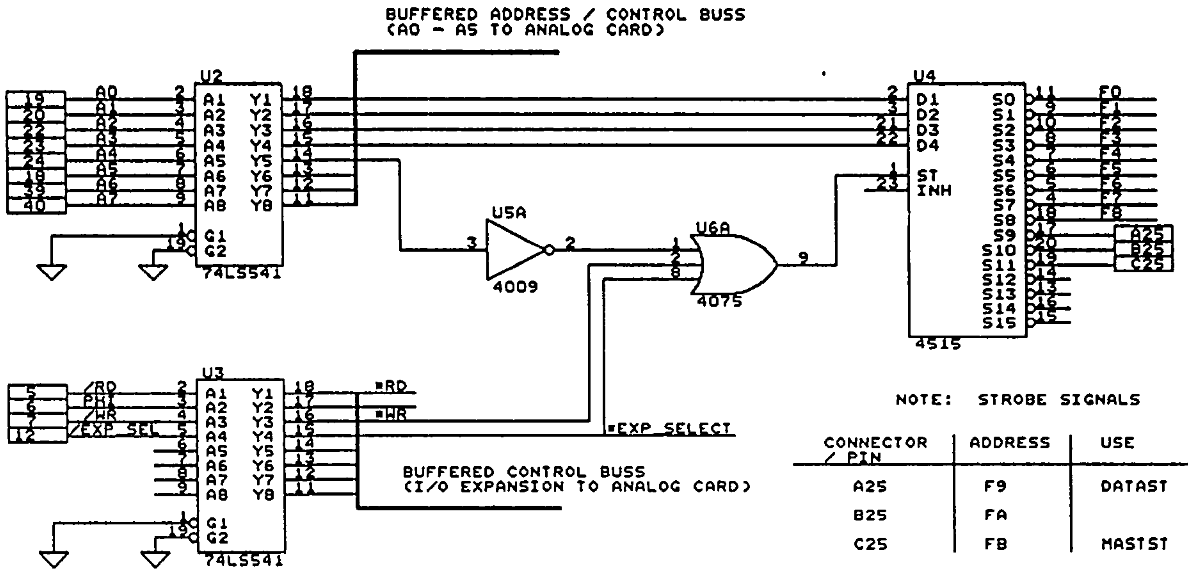
from the TMG (TiMinG) input (pin 3) of the TMS99532 modem determine the time-on and time-off times. For the values selected, with R20 being 4.7 M and C20 being 0.01 μf, the time-on delay is about 32 ms and the time-off delay is about 6 ms.

The timing pulse MCLK (Modem CLocK) is a 4.032 MHz crystal frequency needed by the modem chip in order to provide the right frequencies to meet the Bell 103 300 baud specification. A high speed CMOS gate, U112A (74HC04 inverter), with resistor R11 (10 M) in parallel to bias the gate in its linear region, form the basis of a "high speed CMOS" self oscillating circuit that starts up upon feeling a power on transient. Resistor R12 (1 K) provides an impedance that adds some additional phase shift in conjunction with capacitor C13 (330 pf). This has the effect of cutting out spurious high frequency oscillations isolating the gate output from the crystal network Y110 (4.032 MHz) so that a clean square wave results. The value of R12 is chosen so that is will be roughly equal to the capacitive reactance of C13 at the frequency involved. Capacitors C12 (47 pf) and C13 form the load resistance of the crystal. Most crystals are cut for either 20 or 32 pf load capacitance. Using values larger than this, C12 and C13 swamp out the effects of temperature and supply voltage change on the input and output impedances. Since C13 is so much greater than this, capacitor C11 (33 pf) is placed in series with the crystal to act as the load for the crystal and thus assure an impedance match so that the crystal will not be loaded down. The result is the 4.032 MHz square wave MCLK pulse train.

Finally, the digital levels between the modem and the SB180 are changed from TTL to RS-232C through the use of 1487 and 1488 level shifters. The three signals are the transmit data (TXD), receive data (RXD), and DCD lines.

10.6 Description of the SBC Controlling Latch

Unlike the string modem board, the SB180 computer board as well as the remainder of the interface boards (analog sensing, power module serial communications, and SBC controlling latch) were all mounted in parallel with the length dimension of the SBC electronics cylindrical pressure housing. The first of these interface boards that we will consider is the SBC controlling latch. It is documented in Figure 10.5 below:



NOTE: STROBE SIGNALS

CONNECTOR / PIN	ADDRESS	USE
A25	F9	DATAST
B25	FA	
C25	FB	MASTST

Figure 10.5 The SBC controlling latch interface

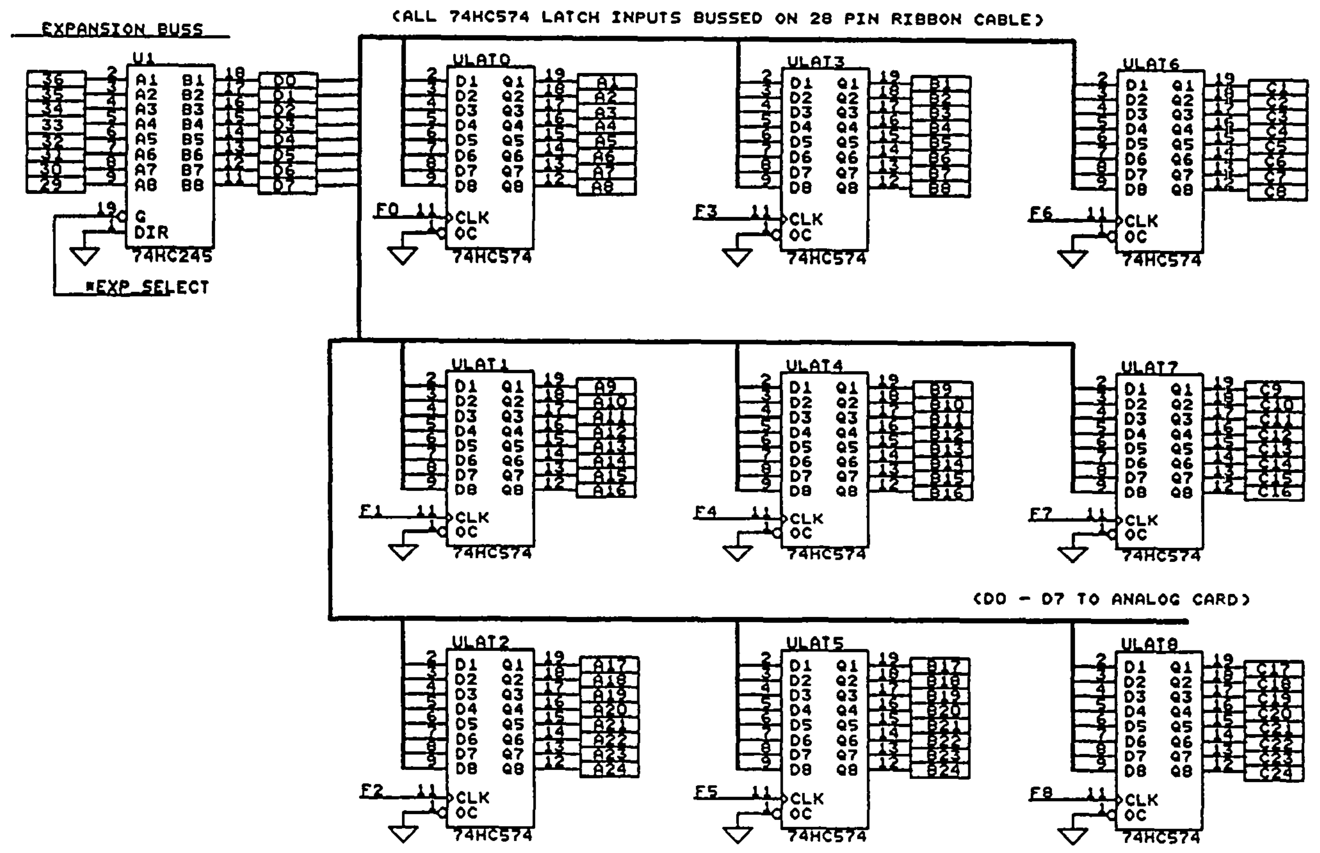


Figure 10.5 (Continued) The SBC controlling latch interface

U1 (74HC245 tristate bidirectional buffer) buffers the data lines D0 through D7 from the expansion buss of the SB180. The parallel output of this chip forms the source of the on board data buss. Additionally, the buffered buss was in turn directed through a ribbon cable to the analog sensor board. A total of nine identical eight bit wide latches (74HC574), ULAT0 through ULAT8 inclusive, share this data buss for input. The latches differ only in their individual data strobe (or selection) lines. The outputs of the latches are grouped into collections of 24 lines each labeled A1 through A24, B1 through B24, and C1 through C24. These are the bits which directly toggle the control lines of the SBC fast electronics.

U2 (74LS541 monodirectional buffer) buffers the address lines A0 through A7 from the expansion buss of the SB180. The parallel output of this chip forms the source of the on board address buss. The last three address lines, A5 through A7 are passed on by way of a ribbon cable to the analog sensing card which is discussed later. A0 through A3 are directed to U4 (4515 4 to 16 decoder) which then breaks out sixteen possible unique addresses. These addresses are found in the mapped memory I/O space of the SB180 at locations 0F0H through 0F8H forming the nine required data strobe lines for the nine latches. Additionally, the next three available addresses, namely 0F9H through 0FBH were brought out and labeled A25, B25, and C25 respectively. A25 was used as the DATAST (DATA STrobe) control of the SBC and C25 was used as the MASTST (MASTer reSeT) control of the SBC. B25 was a spare. As strobes, they directly reflected the level that the software placed them at and were unlatched. The ribbon cables interconnecting the latch (A1-A24, B1-B24, and C1-C24) and strobe (A25, B25, and C25) lines were directed through three 25 pin DB-25 connectors (the A, B, and C connectors) and as such, the line numbers directly reflected the intended signal (ex: pin 17 of the B connector is the latch line B17 or bit 1 of latch 5, etc.).

Table 10.2

Summary of the decoded addresses on the SBC latch control card

Address	Name	Use
Latch control:		
F0	A1-A8	p
F1	A9-A16	o A
F2	A17-A24	r
		t
F3	B1-B8	p
F4	B9-B16	o B
F5	B17-B24	r
		t
F6	C1-C8	p
F7	C9-C16	o C
F8	C17-C24	r
		t
Strobe lines:		
F9	A25 or DATAST	Data strobe
FA	B25	spare strobe line
FB	C25 or MASTST	Master reset strobe

Finally, several control signals were also buffered off of the expansion buss of the SB180. These were the RD (ReaD), PH1 (PHase 1), WR (WRite) and EXP SEL (EXPansion SELect) signals. U3 (74LS541 monodirectional buffer) buffers these control signals. They are all passed on to the analog sensor board, again by way of a ribbon cable. The on board RD, WR, and EXP SEL signals were the only ones used in the control latch logic. Address decoding was accomplished by directing the on board address line A4 through U5A (4009 inverter). In turn, the inverted address

A4, along with the on board WR and EXP SEL lines were all directed through U6A (4076 three input OR) whose output is high active strobe equivalent to addresses 0F0H through 0FFH in write time. This is the strobe used to select the four to 16 decoder U4. The data buffer U1 is selected in RD and EXP SEL time (both signals taken as active low).

10.7 Description of the Analog Card (Serial I/O and Sensors)

The analog card contains two different functional circuits. First, it contains a 16 channel, eight bit D to A circuit for monitoring a collection of analog sensors which continuously probe the environment of the SBC. Second, it contains a fixed 300 baud serial I/O channel. This channel was used to expand the capabilities of the SB-180 so that a third serial port was available to communicate with the power distribution module.

Figure 10.6 documents the auxiliary serial port interface. U12 (74LS541 tristate buffer) buffers the output data from the Universal Asynchronous (serial data) Transmiter / Receiver (UART) U6 (AY-1013) onto the data lines D0 through D7 directly from the (unchanged via the SBC latch card) SB180 computer board. Its parallel input is directed from the output (receive) data port (RR1 through RR8) of the UART. This is a very dumb serial I/O chip. It is controlled by the sense of its input lines rather than by writing anything into a latch. As such, it does not need a computer to program it in an initialization sequence in order to make it run. This was considered preferable in this design since one of the operational goals was to have all serial communications directly open to test with a standard ASCII serial terminal. The parallel data input (transmit lines TR1 through TR8) is wired directly into the raw data buss.

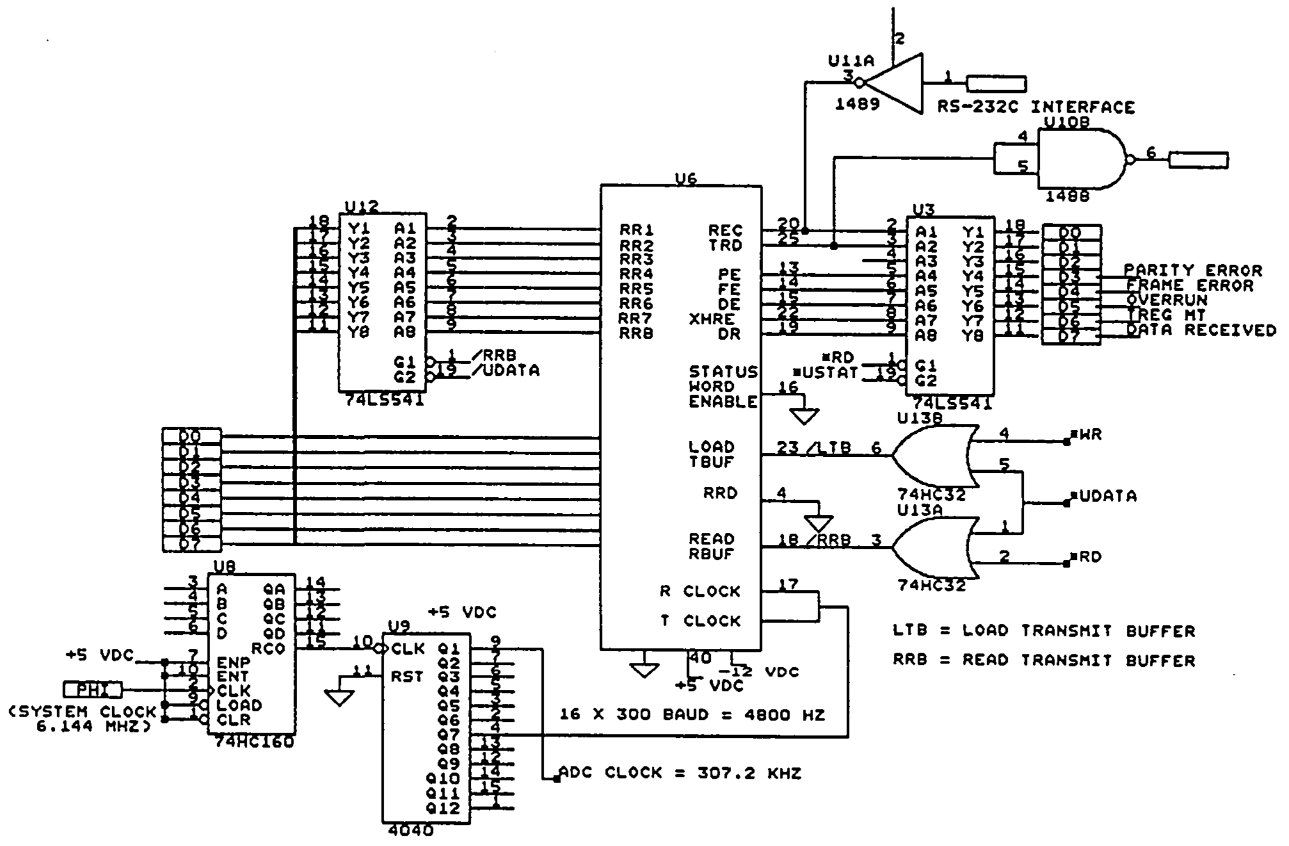


Figure 10.6 The auxiliary (power module) serial port interface

U3 is another tristate buffer (74LS541) which buffers the status lines of the UART onto the data lines D0 through D7. Table 10.3 documents these status lines which are thus made available to the SB-180. For the status word to be available, the input line *SWE (Status Word Enable) is tied low.

Table 10.3

UART status lines

Data line	Status line name	Status indicator
D0	REC	Receive data bit (serial)
D1	TRD	Transmit data bit
D3	PE	Parity error
D4	FE	Framing error
D5	OR	Overrun
D6	TBMT	Transmit buffer empty
D7	DR	Data received

A series of control strobes is available on this board. Their decoded addresses are shown in Table 10.4:

Table 10.4

Control strobes for the analog board

Address	Name	Use
E0	UDATA	UART data
E1	USTAT	UART status
E2	ADC1	A to D conversion
EF	LSTAT	Latch card status

Data is read from the UART in RD (read) time at address 0E0H and written to the UART in WR (write) time at the same address. UART status is read in RD time at address 0E1H. Accordingly, U13 (74HC32 quad two input OR) gates A and B are used to strobe the UART appropriately to either load the transmit buffer during a data write or to read the receive buffer during a data read. The serial communications lines, TRD and REC, which are at TTL level, are level shifted to RS-232C levels through U10 (1488 TTL to RS-232) and U11 (1489 RS-232 to TTL). Note that the TTL level signals are available to the computer at the status latch but that this is not the normal means of conveying serial data information to the computer.

The UART is programmed for a particular serial communications mode by fixing the levels of several special inputs. These inputs are listed in Table 10.5:

Table 10.5

Serial mode control in the UART

Input name	Function
NP	No parity bit
TSB	Stop bit selection
NB2	Number of bits 2
NB1	Number of bits 1

Tying all of these high sets the UART into a mode where parity is inhibited, there are two stop bits, and there are eight data bits. This was the default chosen.

Finally, for the serial communications function, it is necessary to establish the communications rate of 300 baud. This is derived from the system clock of the SB180 and is available as the signal PH1 from the expansion buss. PH1 is the 6.144 Mhz system clock. It is directed to (synchronous 4 bit) decade counter U8 (74HC160) which divides it by ten down to 614.4 KHz. In turn, this is directed to binary counter U9 (4040) whose divide by two output (Q1) is the 307.2 KHz clock (ADC_CLOCK) used to drive the A to D conversion circuit to be described next. The divide by 128 output (Q7) results in a 4800 Hz clock which is 16 times the desired rate of 300 baud. This clock is thus directed to both the transmit and receive clock inputs of the UART. As such, each communications bit time takes 16 clock pulses to complete enabling clear distinction of all transitions in the UART.

Figure 10.7 documents the remaining function of A to D conversion of analog sensor inputs. Central to this is a 16 channel, eight bit A to D converter U7 (ADC0816) which we encountered before in the same capacity in the Optical Module circuitry. In this case, with sensors running all over the inside of the SBC pressure housing, the analog inputs IN0 through IN15 were protected a bit by installing a small RC filter in the form of a 1 K Ω resistor in line with the input and a 0.05 μf capacitor from the line to ground. This way voltage spikes induced in the lines because of their long runs are shunted off to ground. The data lines D0 through D7 are connected to the buffered bus from the SBC latch card. The clock line is the 307.2 KHz clock mentioned above.

Control of the A to D converter is through the use of an external port, U1 (74HC541 tristate octal buffer). It is enabled during $\underline{\text{RD}}$ (read) time at the bit mapped I/O location 0EFH whose strobe is $\underline{\text{LASTAT}}$ (latch status) referred to in Table 10.4. Two status bits are provided on the data buss. A momentary contact switch is wired to a logic high condition at bit D0. This was installed as a simple means of testing the process of reading the A to D status port. The EOC (end of conversion) signal is tied into D1. Thus, a read of the status port indicates whether or not the A to D conversion process is complete. When it is, the microcomputer can then proceed to read the A to D converter parallel data output.

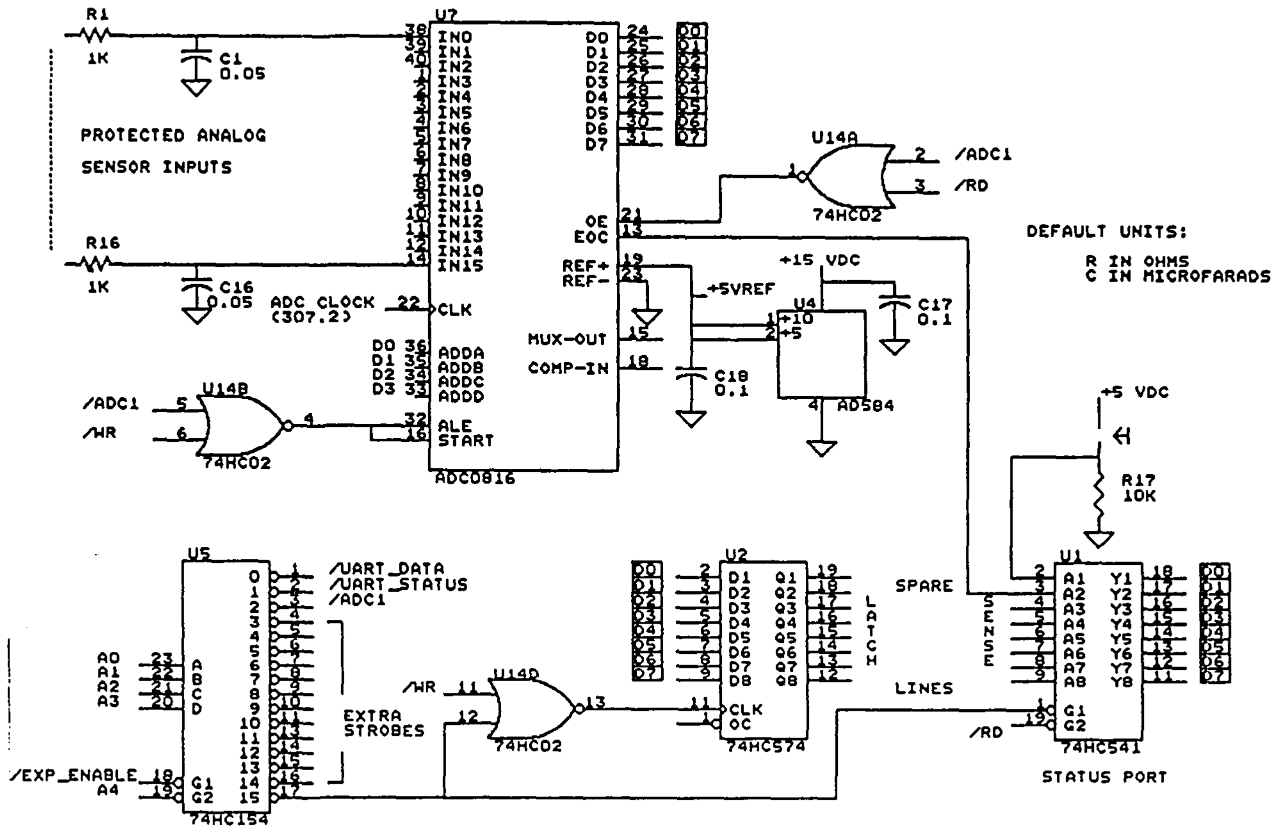


Figure 10.7 The A to D analog sensor interface

The A to D converter starts the conversion process when it receives a START signal made up of the combination in U14B (74HC02 NOR)

START = WR AND ADC1 (10.1)

Similarly, the A to D converter data output is strobed by performing a DREAD (data read) made up of the combination in U14A (74HC02 NOR)

DREAD = RD AND ADC1 (10.2)

The A to D converter includes a 16 channel multiplexer. The internal line COMP-IN (COMParison INput) is the input to the A to D section of the chip. An analog signal in the referenced range of 0 to +5 VDC will be converted to its eight bit digital equivalent. The internal line MUX-OUT (MUltipeXer OUTput) is the output of the 16 channel multiplexer section of the chip. A given channel is selected through the channel selection inputs ADDA through ADDD which are tied directly into the buffered data buss as D0 through D3 inclusive strobed by the START signal. These two lines, COMP-IN and MUX-OUT are tied together to connect the two sections of the chip.

U4 (AD584) is a precision voltage reference configured to provide a reference voltage of +5.000 VDC called 5VREF. This reference is connected directly to the positive reference input +VREF and the +5 VDC power input VCC of the A to D converter. Again, a small 0.1 µf capacitor is connected from the line to ground to shunt off any noise spikes. The A to D converter is a CMOS circuit and as such requires a very small current for it to function. In a similar vein, the negative voltage reference -VREF and ground GND inputs of the A to D converter are tied directly to ground thus completing the power source loop and fixing the A to D conversion reference to +5.000 VDC absolute.

Finally, U5 (74HC154, 4 to 16 line decoder) is the source for the address strobes 0E0H through 0EFH documented in Table 10.2. Address lines A0 through A5 are used in conjunction with the control line $\overline{\text{EXP ENABLE}}$ (EXPansion ENABLE) to produce these strobes. Additionally, a spare data buss latch, U2 (74HC574 octal latch) was provided in $\overline{\text{WR}}$ (write) time at LASTAT but it was not used.

10.8 Description of the Hardware Changes in the SB180 Computer used in the SBC

To complete the discussion of the hardware associated with controlling the SBC, mention must be made of the modifications that were made to a stock SB180. These modifications are minor and as a consequence this section will be brief.

Jumper JP3 is changed as follows: Normally the output pin 6 of U19 which is a 1489 RS-232C to TTL level shifter is connected directly to the $\underline{CTS0}$ input of the HD64180 microcontroller. This is cut and the $\underline{CTS0}$ tied permanently to ground so that is is always selected rather than looking for a handshake.

Jumper JP4 is changed as follows: Normally the output pin 8 of U19 which is a 1489 RS-232C to TTL level shifter is connected directly to the $\underline{DCD0}$ input of the HD64180 microcontroller. This is cut and the $\underline{DCD0}$ tied permanently to ground so that is is always selected rather than looking for a handshake.

To assure a controlled start up of the system, it was decided that it was best to delay the SB180 a short while upon initial power up so that the other electronics would be already powered up when the SB180 first tried to do something with them. This is simply accomplished by replacing the 1 microfarad 100 V electrolytic capacitor C9 with a 220 µf 16 V electrolytic capacitor so that the reset RC time is much longer.

The J6 and J7 expansion buss connectors were used as the source of the address lines A5 through A15 inclusive. A5, A6, and A7 were taken directly from J6. Now, U20 is a socket for a 74LS156 dual 2 to 4 line decoder whose output is connected directly into J7 for the purposes of being able to toggle up to eight different I/O lines. Since control of the SBC required much more than the ability to toggle only eight lines, this socket was left empty but the expansion lines were used to bring out the additional address lines A8 through A15 as defined in Table 10.6 below. These were all wired over from the nearby 4.7 K pullup resistor pack SIP3.

Table 10.6

The J6 expansion line definitions

J6 pin number	SIP3 pin number	Address line
1	3	A8
2	4	A9
3	6	A10
4	8	A11
5	2	A12
6	5	A13
7	7	A14
8	9	A15

Finally, the power supply connector J7 was modified to stick upright from the board so that it could be easily connected and disconnected in the very cramped housing area for the SB180 in the SBC. Dummy pins were added so that the connection could be made only in one way preventing accidental destruction of the computer through application of wrong voltages.

10.9 Description of the SBC Controlling Program MERLIN

The second half of this chapter documents in some detail the SBC control program called MERLIN. The program is the brains of the SBC. It performs three basic functions. First, it acts as a traffic cop for all command and control communications in the SPS. Second, it decodes and executes commands directly intended for the SBC hardware. Third, it has a lot of exception code written to get around inconsistencies found in integrating major subsystems of the SPS. The SB180 computer comes equipped with a ROM based monitor that was documented above. This monitor has many desirable features for the control and testing of the hardware within the SBC. Further, the nature of the SBC environment required that the control computer have a ROM based program. Since the SB180 ROM monitor has provisions for jumping out of the ROM monitor into a user defined routine, it was decided that the best way to maintain control of the SB180 was to make SBC control program a routine under the ROM monitor. Thus, operationally, when the system first starts up, the ROM monitor gets control of the SB180. Later, after everything checks out, the command within the ROM monitor for jumping to a user routine is executed and then the SBC control program takes over.

10.10 The SB180 Development System

The operating system[3,4] that is supplied with the SB180 is important to the system developer for two reasons. One, it is the system within which all software writing, compiling, linking, and archiving takes place. Two, it is this one and the same system which plays the role of a ROM simulator in the process of debugging the program. In this way, the program can be

---

[3] see Conn, Richard, ZCPR3 The Manual

[4] also Ciarcia, Steven, Build the SB180 Single-Board Computer, Part 2: The Software

dynamically changed and tested without having to go through the tedious task of burning and erasing ROM's with each iteration.

The name of this operating system is the Z-System (a product of Echelon, Inc.). It is considered to consist of two main machine independent parts, ZCPR3 and ZRDOS. This system is a Z-80 based CP/M (Control Program / Microprocessor, an early standard microcomputer operating system put out by Digital Research Corporation[5]) upward compatible operating system. As such, since it can run CP/M programs, it runs a large collection of existent software. This means that development with this hardware was not held up in any way in having to wait for the software. As you will see, the overall development system was thus readily enhanced through the use of software tools developed elsewhere. Additionally, as part of the development environment, close to 100 integrated utilities came with the Z-System permitting flexible and intelligent extension of the operating system.

Good software techniques require the separation of functions in the form of layers surrounding a central execution core. The Z-system is no exception. The outermost layer is ZCPR3 (Zilog CP/M Replacement number 3). This is the command processor. The next interior layer is ZRDOS (Zilog Replacement Disk Operation System). This is the core of the operating system. It creates the standard virtual machine to which applications programs connect and run. The next layer is the BIOS (Basic I/O System). This is the software glue that connects the actual hardware being used to the standard calls of ZRDOS. Thus, ZCPR3, the utilities, and ZRDOS are machine independent (except that the instruction set must be a Z-80 or something upward compatible) while the BIOS is where all of the machine dependencies are collected together and must be rewritten for each new type of computer.

Let us first look at the outer most layer, namely the command processor ZCPR3. This is the user interface to the rest of the operating

[5] Digital Research, CP/M 2.0 User's Guide

system. As mentioned, associated with it is a collection of about 100 utilities specifically written to take advantage of the special features of the ZCPR3 environment. One of the really nice things about this is that often, when one wants to write a system program, the utilities already perform all or most of the functions desired. From this point of view, they can be viewed as filters in which part of the output of one may become the input of another, etc. This approach to computer programming was first popularized by the UNIX[6] operating system. It views programs as a string of filters and typically one's entire program is simply the command line of "piped" filters. ZCPR3 has a somewhat similar philosophy allowing one to string together utilities or other routines in meaningful ways treating the combination as a new program.

Intrinsic to ZCPR3 are six system segments. Upon booting the system, six separate memory areas are reserved and loaded. These six segments form the basis that makes the Z-System so much more useful than CP/M. Each segment either adds features or contains information that properly written programs can make use of in an integrated way. To understand how this works, it is best to look at each of these segments in turn. In what follows, the bracketed [ ] hexadecimal number is the location in SB180 memory of the segment.

ENV - ENVironmental descriptor [0FE00H-0FEFFH]: There are many ways to configure a system. As such, this segment contains the description of how this particular ZCPR3 implementation is configured.

NDR - Named DiRectory [0FC00H-0FCFFH]: This segment assigns symbolic names to disk drives and user areas. For example, let us say that disk drive B, user area 7, contained a collection of filters and files associated with an accounting application. Instead of having to remember to refer to B7 in order to access these files, one can assign the name ACCOUNT to this area.

[6] for example, see Kernighan and Plauger, Software Tools

RCP - Resident Command Package [0F200H-0F9FFH]: This is a collection of subroutines extending the intrinsic commands of the operating system. An intrinsic command is one which resides in memory but not in the area set aside for application programs (called the Transient Program Area or TPA). Examples of ZCPR3 intrinsic commands are GO, SAVE, GET, and JUMP. These are all ZCPR3 command processor level functions. Examples of extended commands found in the RCP are CP, ERA, TYPE, LIST, PEEK, POKE, PROT, and REN. One has to make choices because the RCP is limited to 2 kilobytes of memory. Thus, depending upon the environment, different commands may be loaded into the RCP.

FCP - Flow Control Package [0FA00H-0FBFFH]: This is unique to ZCPR3. It adds structured conditional testing to the operating system level commands. This is especially useful in batch operations. The keywords are self explanatory being IF, ELSE, and FI (an "end if"). Nesting of conditional tests is possible up to eight layers deep. Typical conditional tests are shown in Table 10.7 below:

Table 10.7

Conditional tests for flow control

Name	Test Function
Negation	Reverses sense of logical condition
True	Tests if logically true
False	Tests if logically false
EMpty	Tests if file is empty
ERror	Tests program error code byte
EXist	Tests for existence of a file
INput	Tests for user input of character 'T'
NUll	Tests if second argument is not specified
n	Tests if indicated register contains the specified value n
WHeel	Tests if wheel byte is set or not
TCap	Tests if TCAP contains a terminal definition or not
fcb1=fcb2	Tests two file control blocks to see if they contain the same values

IOP - Input / Output Package [0EC00H-OF1FFH]: The code in this segment acts as a traffic cop routing I/O to and from peripheral devices. An example is the printer spooler supplied with the SB180.

TCAP - Terminal CAPabilities [0FE80H-0FEFFH]: This segment is actually part of the ENV segment although it can be loaded independently. It describes the characteristics of the terminal connected to the computer, especially the control strings for clearing the screen, cursor addressing, highlight on/off, and the arrow keys.

Besides the six system segments, there are several other concepts associated with the ZCPR3 command processor that are important to consider. These are the path, the wheel, utilities, shells, and aliases.

The concept of the path is that there should be some means by which ZCPR3 can search other directories (disk drives and user areas) for a file should the file not reside in the current user area. Small computer operating systems typically have not had this ability. ZCPR3 provides a means by which to specify the path so that up to five levels deep may be searched. Typically one directory may contain help files, another may contain system utilities, another may contain commonly used programs, etc. and the path is the means by which these can be systematically and rapidly searched rather than forcing the operator to explicitly remember where a file is.

The wheel (located at 0FDFFH) is a single byte protection system for ZCPR3. If it is zero (blank), the wheel is considered reset (off). If it is non-zero, the wheel is considered set (on). All intrinsic ZCPR3 commands check the wheel for user privilege to execute the requested command. Similarly, utilities can be written to check the wheel. Programs that manipulate the wheel require a password to operate.

The role of utilities has been mentioned before. Basically, utilities are transient programs that do useful system level work. Typically they access the ENV, TCAP, and NDR in working with the system. The ZCPR3 utilities have all been written with a common approach in mind. They make great use of the system segments, the path, and the wheel. One of the things that they have in common is a built in help screen. This is a brief example of the syntax of the utility and how it is used. The help screen is invoked by typing on the command level the name of the utility followed immediately by "//". For example, to get the help screen of a utility called LDR, one types "LDR//".

Shells are another concept borrowed from UNIX. As mentioned before, the software in a computer can often be viewed like the layers of an onion with the outer layers passing calls to the inner layers and the inner layers passing information concerning the calls back to the outer layers. This way, machine dependencies can be kept to a minimum and programmers normally need not worry about the particular hardware that their application is running on. In the case of the Z-System one might stack the layers as follows in Figure 10.8:

application program

shells...

ZCPR3 command processor

ZRDOS disk operating system

BIOS machine dependent I/O

HD64180 instruction set

Figure 10.8 The layers of the Z-System

Normally when one is running a program what is happening is that the ability of the command processor to "run" an executable file is being invoked. In other words the application program in some sense is under the control of the command processor. At least the program should return to the command processor in a non-destructive manner. Now, let us say, for the sake of argument, that one had an application program which did not abide by the syntax of the command processor. Perhaps it uses a different convention for naming disk files or the keyword order is different. One could still run such a program should a piece of software exist that translates between the two different sets of conventions. Another possibility is the desire to use one key to generate a whole command string. Such software sits in a layer between the command processor and the application program and is called a shell. This can be very useful in turnkey applications where one does not want to have to teach the operator a lot of things about the system just to run an application. In fact, the concept is extendable and shells can be nested.

Finally, there is the concept of an alias. This is the way in which ZCPR3 is able to link up a series of existing routines and turn them into a new program in its own right. This concept of not reinventing the wheel is borrowed straight from UNIX. An alias is created by invoking the alias utility which basically takes as a command line a string of commands to be placed in the multiple command line buffer and executed. Parameters may be passed and aliases nested allowing creation of very powerful commands.

So far, the emphasis here in outlining the SB180 development system has been on the ZCPR3 command processor and associated concepts such as utilities. This is because this is the level the operator using the Z-System for program development is normally going to be dealing with. For the sake of completeness I will mention briefly some of the characteristics of ZRDOS and the BIOS.

The next layer in is ZRDOS (Zilog Replacement Disk Operating System). In the SB180 it resides in the segment bounded by 0CC00H-0D9FFH. As the disk operating system, it is the core of the Z-System defining a standard virtual machine for the programmer. It is a replacement for the analogous CP/M BDOS (Basic Disk Operating System) and is upward compatible with the CP/M 2.2 standard. This includes all 39 functions found in BDOS plus four others unique to the Z-System. The visible differences with CP/M is that there is no longer the need to type <CTRL>C to log a new disk onto the system and there are improved error messages. This is a great help in setting up turnkey applications.

Finally, the BIOS (Basic Input / Output System) resides in the segment bounded by 0DA00H-0EBFFH. It is the interface between the operating system (ZRDOS) and the specific hardware (the SB180 with its Hitachi HD64180). Most importantly for the SB180 is that it knows about the enlarged memory space of 256 kilobytes of RAM. 192 kilobytes are excess beyond the range of CP/M and so are used by the BIOS as a RAM disk drive. This is accomplished by dedicating one of the DMAC channels to this function. A RAM disk greatly speeds up disk intensive operations by making the "access" to and from the "disk" actually the speed of reading and writing to RAM. No hard disk can ever operate that fast. Once the disk intensive activity is complete, then the results can be saved on a real disk.

The second part of the development system are the language tools used to write the program in. Primarily the program was written in a CP/M implementation of "C". C is a language that is sometimes greatly misunderstood. Although the textbooks do not seem to mention this, the best way to look at C is to look at it as a universal assembly language. If you think about it that is a neat trick because assemblers, by their very nature of working with different instruction sets have to be different from one another. However, if one takes a careful look at extensive assembly language code, one finds that only ten percent of what is actually written is truly specific to the hardware involved. The rest only becomes specific because the programmer wrote the program in the assembler associated with that machine. It is often possible with different compilers to restrict machine dependencies to the minimal assembly code needed and then to call those assembly routines from some (hopefully) machine independent compiler.

Fine, but C goes even further than that. C is designed to bang information right down to the bit. As such, machine dependent assembly code is even smaller than the usual compiler linked to assembler code approach. Additionally, for applications where execution speed is critical, C typically generates code close to being as tight as highly tuned assembly code (and takes a lot less time to write and debug). C is a very stripped down language so the overhead associated with it is quite minimal. There are not many functions built into the language anymore than there are in the typical assembly language. The exception to this is that on larger systems, C often can be alternatively compiled to know about formula translation and execution like FORTRAN.

Finally, C is very portable not only because of its intrinsic machine independence but because it has been almost a religion with programmers not to change or extend C. As a consequence it does not suffer from "versionitis" as most other languages do.

With C's stripped down ability to manipulate the hardware, it was the ideal choice for programming the SB180. Fortunately a number of C compilers exist under CP/M[7]. The one that was used is a subset of the full specification[8] called C/80 (a product of The Software Toolworks[9]). Basically it is limited to integer arithmetic which is more than enough for the task at hand.

Fundamental to the approach of C (and of UNIX) is the idea of breaking big projects down into smaller pieces or modules. The modules are all then made to work by themselves and then combined to obtain a working system. This was exactly what was done with the SBC controlling program, MERLIN. The hardware dependencies were isolated into their own modules as embedded assembly code which is a feature of this compiler.

Another aspect of C which is worth mentioning is its preprocessor. This is a device which, among other things, permits macros to be attached to the body of code as well as to conveniently DEFINE constants much as

---

[7] see Cain, Ron, A Small C Compiler for the 8080's and Cain, Ron, A Runtime Library for the Small c Compiler

[8] see Kernighan and Ritchie, The C Programming Language. This is THE book.

[9] Bilofsky, Walt, C/80 Small C Compiler.

one makes equate (EQU) assignments in assembly language. Similarly, global variables may be defined. In general, numbers should never be written directly into a program but instead given a symbolic name in one location so that if the value is to be changed it can be done so neatly and cleanly in only one place. Typically, such collections are made in a series of "header" files which through the preprocessor command INCLUDE are drawn into the source code file during compilation. This is all very similar to some of the pseudo opcodes found in assembly language.

The output of the C/80 compiler is raw assembly code. This is nice because it gives one an additional handle in the form of tuning the code should that be necessary. A number of switches exist permitting alternative code generation for different assemblers.

The assembler used was M80[10] ("macro" 80, a product of Microsoft). It is a fairly standard macro assembler following a macro calling convention quite similar to Macro 11 (on Digital Equipment Corporation PDP-11's). Thus, it is quite different than that followed by INTEL, as mentioned before, in terms of the ASM-51 cross assembler used to program the 8051 microcontroller family which UHPS is entirely written in. The output of M80 is a relocatable binary file which must be linked together with other such files. Finally, the resultant linked file must be loaded into memory at absolute addresses before the program can be executed.

Typically in the old CP/M environment and not necessarily different in the Z-System environment, linking and loading have been basically combined into one separate operation. The basic reason for this is that once linked, the standard CP/M utility DDT (Dynamic Debugging Tool) is used to either save the image to disk or to make a controlled run of it. The nature of most eight bit microcomputer instruction sets is such that all executable code is what is called absolute. This means that the program must reside in one and only one specific place in memory. References within the code point

[10] see Microsoft MACRO-80 ASSEMBLER Software Reference Manual

to absolute addresses. Larger machines, particularly those which are multiuser have executable code which is relocatable. Their references are relative.

Because of this, it means that at the time of linking, all relocatable binary modules, plus the linker-loader, plus the symbol table for resolving references, plus the resultant absolute executable code must all reside in the TPA (Transient Program Area or heap memory available to executing programs) at one time. During the course of writing MERLIN, I ran out of room. The problem was eventually resolved by finding and using SLRNK+ (SuperLinker Plus - a product of SLR Systems[11]). This is a non-loading linker for Z80 based CP/M systems. It is limited to execution on Z80 compatible systems (which in the case of the HD64180 is no limitation at all) because it makes heavy use of the additional instructions and registers of the Z80 over the original 8080. The result is that it is very fast. Instead of using the TPA to hold all of the files, it makes heavy use of the disk drive. Thus it is able to link very large systems. Better yet, since the SB180 has such a large RAM disk, by feeding everything to the RAM disk, the SLRNK+ linker becomes lightning fast.

This just about concludes the discussion on the SB180 development system. There is one thing left that should be emphasized. Ultimate debugging of the program requires testing the program with the actual hardware that is going to be controlled. Even if the program should run as designed, the actual manipulation of things results in insight into how things ought to be changed and hopefully improved. Thus, by using a ROM simulator in terms of the ZRDOS disk operating system, parallel development of an improved version of MERLIN was possible while others exercised an older version in tuning up the ship board control software. One did not have to constantly disassemble the SBC electronics to get to the imbedded SB180 to make a program change because it was connected by way of a ribbon cable to an external floppy disk drive.

[11] SLR Systems, SLRNK+ Superlinker Plus User's Guide

10.11 Software Objectives

Before going on to describe MERLIN in detail, it is best to first outline the objectives of the program. Roughly speaking, MERLIN was to be the central control program for the SPS. Commands were to come down from the ship (either directly from an operator on a terminal or indirectly from the ship board computer). These were to be parsed by MERLIN and basically sorted into one of three categories:

The first category is identifying a command specifically for a single module on the string or for the power distribution module. In all such cases, MERLIN passes the command on to the appropriate channel (either the STR or PWR ports) in a transparent manner. The protocol demands that such commands must make a reply which MERLIN should receive and then relay back up along the cable both by way of the 300 baud electrical connection (the CAB port) and the high speed fiber optics connection (the OPT port). The ports involved are all summarized below in Table 10.8.

Table 10.8

SBC control computer ports

Port Name	Description
CAB	Cable port (SB-180 CONsole port)
STR	String port (SB-180 AUXiliary port)
PWR	Power module port (analog card)
OPT	Optical port (latch card - 23 word, output only)

Now, there is a danger that the system might hang waiting upon a reply message that may never come. Thus, there must be a time out mechanism that will generate a time out error reply should the expected reply not occur.

The second category is identifying a command that is specifically for the SBC. Typically this means that something is to be written into its latch. The latch is a "write only" memory and as such, in order to be able to conveniently keep track of the current settings MERLIN should maintain a table which mirrors those settings.

The third and final category is sort of the catch all for everything else. As system integration progressed it became more and more obvious that it was good that such a dynamic environment as the SB180 (in contrast to the 8051) was being used to control the SBC. Because the system was sufficiently dynamic, it was possible to write significant exception code to integrate everything between the string on the one end and the ship board multitasking real time operating system (Digital Equipment Corporation's RT11) on the other end. To cleanly handle the task of controlling the SPS and intercepting the replies under RT11 it was felt that it was not a good idea to burden the ship board computer with too many exceptions.

One of the major exceptions to be handled was the case of an "all call" to either all of the optical modules or to all of the calibration modules or to both. Obviously, the optical modules and calibration modules could not send replies under such a situation because more than one module would be trying to communicate on the string at one time and this would shut down communications. The ship board task demanded some kind of reply message before it would send out another command. As such, the SB180 needed to generate a "fake" reply message for the all calls.

Another, somewhat similar exception resulted because the environmental module was not developed to communicate on the string in accordance with the agreed upon protocol. It did not send reply messages so again the SB180 was used to generate them.

Finally, there is the possibility that a command does not parse out correctly either because what was sent was wrong (command error), a non-existent or inconsistent device was requested (device error), or the command format was incorrect (format error). The exception handling functions for such error conditions also belong to this catch all category.

10.12 MERLIN

This final section is devoted to documenting the SBC control program MERLIN in some detail. The governing philosophy of the program and its global data structures will be fully explained. Then a summary of all of its component routines, broken down into categories, will be made. Next, their interconnections will be briefly documented. Finally, some of the actual routines will be flow charted as representative examples of the different categories of routines.

The overriding philosophy of MERLIN is one of continuous polling of all of the potential input ports. As Table 10.8 shows, there are three such ports, namely CAB, STR, and PWR. Since all three ports communicate at only 300 baud or about one byte every 333 ms or so, then in a worse case situation of all three ports sending a byte at the same time, the system needs to look at each port only once every 333 / 3 = 111 ms. With a 6 MHz computer, that means that 666,000 clock cycles are available between polls. Easily 100,000 instructions or more can be executed in that time and thus there is no need for a more complicated system involving interrupts and multitasking.

All character input is first trapped on a byte by byte basis and placed in a single byte input buffer. There is one such buffer for each of the three input ports (inbuf). Every time a new byte is moved into one of the buffers, a flag bit is set (inflag). Later, during the polling, other routines which need this input information check the appropriate input flag and if it is set, they fetch the input byte from its buffer and clear its corresponding flag.

The input port that is given the most consideration is CAB. Here commands coming down from the ship are received. A preliminary parse is made of the command as it comes in to see if it is destined for the string, the power module, or the SBC itself. If it is not for the SBC, then the command is relayed transparently to the appropriate port. Commands directed to either the power module or to individual optical or calibration modules on the string require a reply message. MERLIN looks for such a message and while polling it keeps track of some time out period on the order of five seconds within which time a reply message from the addressed module should be received. If none is available, then a time out error message is sent.

Should the command be for the SBC, then it is parsed by the default parser on a byte by byte basis and either a command is extracted or an error message reply generated. Similarly, if the command fits one of a number of exception conditions such as those mentioned above, then it is parsed by one of the exception parsers. When there is a valid SBC command, then the appropriate driver is invoked and the command is executed. In turn, the SBC generates an appropriate reply and sends it back up the cable over both the slow speed path (CAB) and the high speed path (OPT).

The first files of importance for MERLIN are the C compiler headings. These are listed in Table 10.9 below:

Table 10.9

C compiler headings

File name	Contents
SBCDEF.H	Defines global constants
SBCGLOB.H	Defines global variables
SBCMON.H	Defines ROM monitor constants for MERLIN program

As one can see, they form convenient places for collecting all of the common definitions found in three different categories. SBCDEF.H defines the global constants. These include both the symbolic names of the ports (like CAB or OPT), of the different instruments (like OM3 or POWER), of the mask bit patterns, and of special characters or conditions. SBCGLOB.H contains the definitions for all of the global variables and data structures. These are listed in detail in Table 10.10 below. Finally, SBCMON contains the definitions used in the ROM monitor that must be used in an identical manner in the control program MERLIN.

Table 10.10

The global data structures

Name	Description
inbuf	Buffer - 3 integer bytes to contain the most recent input characters from the three input ports CAB, STR, and PWR respectively
inflg	Flag - 1 integer byte companion to inbuf. If corresponding bit is set it means that there is a byte in inbuf waiting to be read by the parsers.
time	Flag - 1 integer byte indicates if TRUE that a time out is underway on either the STR or PWR ports. This is cleared when a character is received on either port unless the time out period is up.
poll	Flag - 1 integer byte indicates if TRUE that parsing is to be done by EVMPOL instead of the default CMDPOLL.
talk	Flag - 1 integer byte indicates if TRUE that parsing is to be done by ALLPOLL instead of the default CMDPOLL.
sbcrun	Flag - 1 integer byte indicates if TRUE that the routine MERLIN is to be still run. Otherwise, MERLIN exits to the ROM monitor of the SB180.
dev	Register - 1 integer byte holds the identity of the current module the command is destined for.

Table 10.10 (Continued)

The global data structures

Name	Description
cmd	Buffer - 8 integer bytes hold the current command string from the CAB port.
bytnum	Counter - 1 integer byte holds the current count of command bytes received so far while a command is being parsed. This acts as an index to bytes in the command string.
count	Counter - 1 integer byte holds the current count of timeout periods. Used in the timeout routines for message replies from the STR and PWR ports.
table	Table - 9 character bytes holds the current latch settings of the SBC fast electronics. This way, if desired, the current settings can be read.
latch	Latch - 9 character bytes of the SBC which are "write only". Routines write bit patterns to set latch through table so that there is a history of the settings available to MERLIN.
command	Register - 2 character bytes used in debugging to confirm the action of the parsing routines to correctly parse out the command.

Altogether, MERLIN is made up of 49 separate routines each occupying a separate file A summary description of these routines divided into eight basic classes follows directly in Table 10.11.

Table 10.11

Summary of MERLIN Routines

Class Name	Description
Main:	
main	Main program (MERLIN)
inpoll	Parse the command string for a valid command - also, handle exceptions
timpoll	Parse for a STRING command, if found, initiate time out for reply
allpoll	Parse for an "all call" STRING command - handle exception
evmpoll handle	Parse for an environmental module command - exception
nextbyte	Place next byte in command buffer
cmdserv	Service a complete command, else call error routine
allserv	Exception code for an "all call" command
evmserv	Exception code for an environmental module command
Commands:	
coinpat	Coincidence pattern
rawsel	Raw select
stkhgt	Stack height
source	Source on/off
sorsel	Source select
xtrbit	Extra bit (drive spare lines)
readad	Read an analog (A to D) channel
mreset	Master reset

Table 10.11 (Continued)

Summary of MERLIN Routines

Class Name	Description
I/O Routines:	
datlat	Move SBC optical path data from internal table to external latch
tablat	Move SBC latch control data from internal table to external latch
outlatch	Output a byte to a latch port
inport	Read a byte from specified port
outport	Write a byte to specified port
Error Routines:	
deverr	Device error reply
forerr	Format error reply
cmderr	Command error reply
timerr	String timeout error reply
Communications:	
msgreply	Output message on CAB and OPT
allecho	Output byte on CAB, STR, PWR, & OPT
upecho	Output byte on CAB and OPT
downecho	Output byte on PWR and STR
evmmsg	Output fake environmental module message on CAB and OPT
allmsg	Output fake "all call" replies on CAB and OPT

Table 10.11 (Continued)

Summary of MERLIN Routines

Class Name	Description
Type Checking and Conversion:	
ascbin	Convert ASCII character to binary
binhex	Convert byte sized binary numbers to two byte hex equivalent
nibhex	Convert four bit binary nibble into ASCII hex character
ishex	Test character if it's an ASCII hex number
Initialization and Control:	
init	Initialize CAB and STR at 300 baud
deinit	Return to SB-180 ROM monitor
delay	Delay one millisecond
Device Drivers:	
atod	Convert specified channel analog input to digital value
incab	Input a character from CAB
instr	Input a character from STR
inpwr	Input a character from PWR
outcab	Output a character to CAB
outstr	Output a character to STR
outpwr	Output a character to PWR
outopt	Output a character to OPT

To get some idea as to the complexity of the relationships between the routines, a call tree for the program MERLIN follows in Table 10.12. Modern structured languages like C pass parameters by value on a stack rather than by location in memory. Every time a routine is called, the return address and other context must be placed on the stack. Thus a call tree is a document which shows who calls whom and to how deep the stack may actually go. Typically, in a call tree, once a routine is documented all the way in its calls to some terminal hardware driver or whatever which can not or does not call any deeper, then when that routine is called yet again, only it and none of its subsequent calls is documented because that path has already been documented thoroughly.

Table 10.12

Call tree for program MERLIN

```
init
clear
inpoll
        incab
        allecho
                upecho
                        outcab
                        outopt
                                datlat
                downecho
                        outstr
                        outpwr
                instr
                inpwr
                timerr
                        upecho
```

Table 10.12 (Continued)

Call tree for program MERLIN

```
cmdpoll
    evmpoll
        nextbyte
            forerr
                msgreply
                    upecho
                upecho
                downecho
                    outstr
                    outpwr
        evmserv
            evmmsg
                upecho
            cmderr
                msgreply
                upecho
                downecho
        ishex
    allpoll
        nextbyte
        allserv
            allmsg
                upecho

            cmderr
        forerr
```

Table 10.12 (Continued)

Call tree for program MERLIN

```
timpoll
        nextbyte
deverr
        msgreply
        upecho
        downecho
cmdserv
        coinpat
                ishex
                ascbin
                tablat
                        outlatch
                cmderr
        cmderr
        rawsel
                ishex
                ascbin
                tablat
                cmderr
        stkhgt
                ishex
                ascbin
                tablat
                cmderr
        source
                ascbin
                tablat
                cmderr
```

Table 10.12 (Continued)

Call tree for program MERLIN

```
                sorsel
                        ascbin
                        tablat
                        cmderr
                xtrbit
                        ascbin
                        tablat
                        cmderr
                readad
                        ishex
                        ascbin
                        atod
                        binhex
                                nibhex
                msgreply
                        upecho
                        cmderr
                mreset
                        outlatch
        forerr
upecho
deinit
```

Finally, let us consider example flow charts for some of the routines. The top level of any C program is given the reserve name "main". Figure 10.9 documents the main routine of MERLIN:

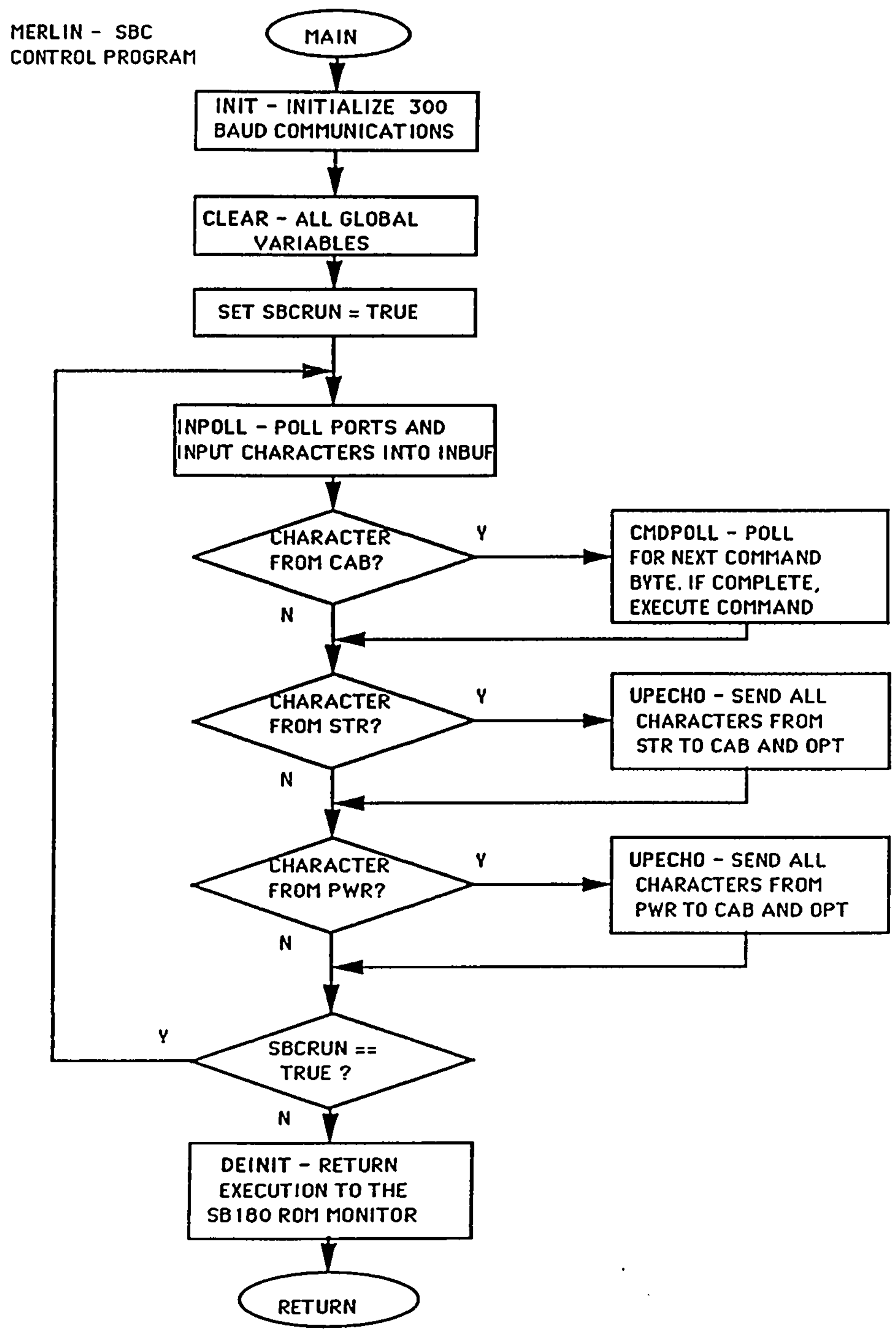

MERLIN - SBC
CONTROL PROGRAM
MAIN
INIT - INITIALIZE 300 BAUD COMMUNICATIONS
CLEAR - ALL GLOBAL VARIABLES
SET SBCRUN = TRUE
INPOLL - POLL PORTS AND INPUT CHARACTERS INTO INBUF
CHARACTER FROM CAB?
Y
CMDPOLL - POLL FOR NEXT COMMAND BYTE. IF COMPLETE, EXECUTE COMMAND
N
CHARACTER FROM STR?
Y
UPECHO - SEND ALL CHARACTERS FROM STR TO CAB AND OPT
N
CHARACTER FROM PWR?
Y
UPECHO - SEND ALL CHARACTERS FROM PWR TO CAB AND OPT
N
Y
SBCRUN == TRUE ?
N
DEINIT - RETURN EXECUTION TO THE SB180 ROM MONITOR
RETURN

Figure 10.9 The main routine of MERLIN

Inpoll is one of the major routines of MERLIN from the point of view of MERLIN's direct mission, namely monitor the communications of three serial input ports and then to act on the information extracted. Inpoll polls all three ports for a character.

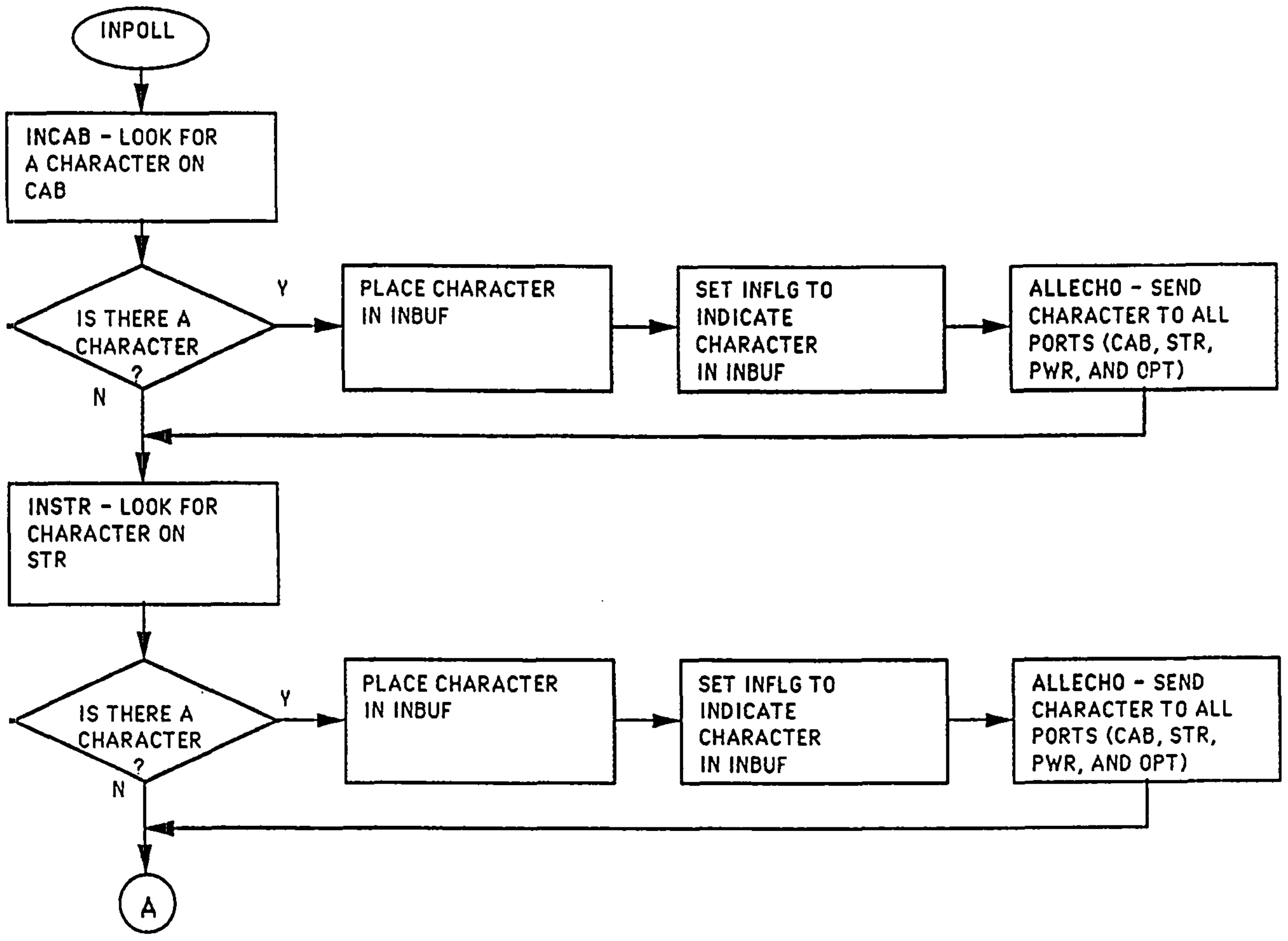

INPOLL
INCAB - LOOK FOR A CHARACTER ON CAB
IS THERE A CHARACTER ?
Y
N
PLACE CHARACTER IN INBUF
SET INFLG TO INDICATE CHARACTER IN INBUF
ALLECHO - SEND CHARACTER TO ALL PORTS (CAB, STR, PWR, AND OPT)
INSTR - LOOK FOR CHARACTER ON STR
IS THERE A CHARACTER ?
Y
N
PLACE CHARACTER IN INBUF
SET INFLG TO INDICATE CHARACTER IN INBUF
ALLECHO - SEND CHARACTER TO ALL PORTS (CAB, STR, PWR, AND OPT)
A

Figure 10.10 The main level routine inpoll

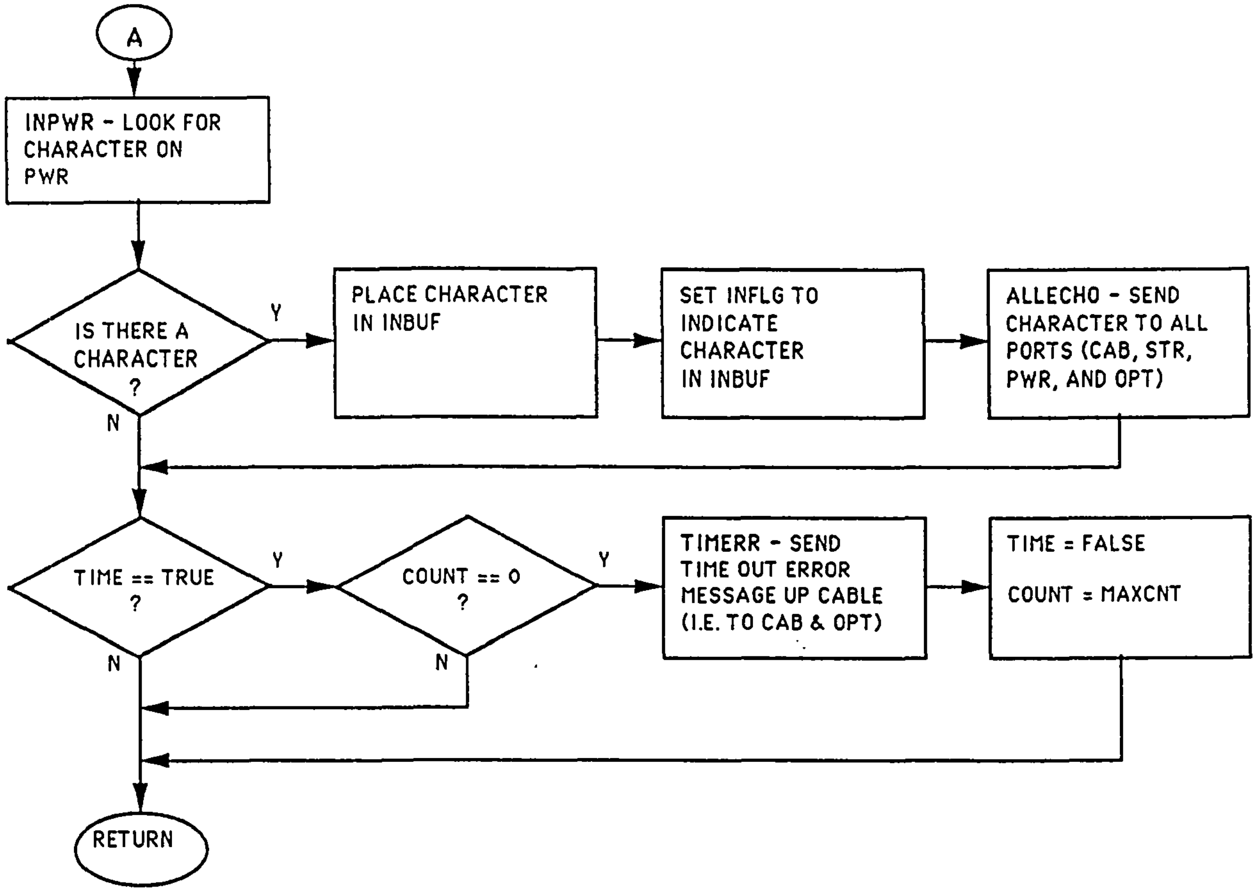

A
INPWR - LOOK FOR CHARACTER ON PWR
IS THERE A CHARACTER ?
Y
N
PLACE CHARACTER IN INBUF
SET INFLG TO INDICATE CHARACTER IN INBUF
ALLECHO - SEND CHARACTER TO ALL PORTS (CAB, STR, PWR, AND OPT)
TIME == TRUE ?
Y
N
COUNT == 0 ?
Y
N
TIMERR - SEND TIME OUT ERROR MESSAGE UP CABLE (I.E. TO CAB & OPT)
TIME = FALSE
COUNT = MAXCNT
RETURN

Figure 10.10 (Continued) The main level routine inpoll

Cmdpoll is another major routine. It is the default command parser for serial communications from the CAB port. It parses the serial stream on a byte by byte basis and either completes the job of parsing or else it invokes some exception code depending upon what it finds.

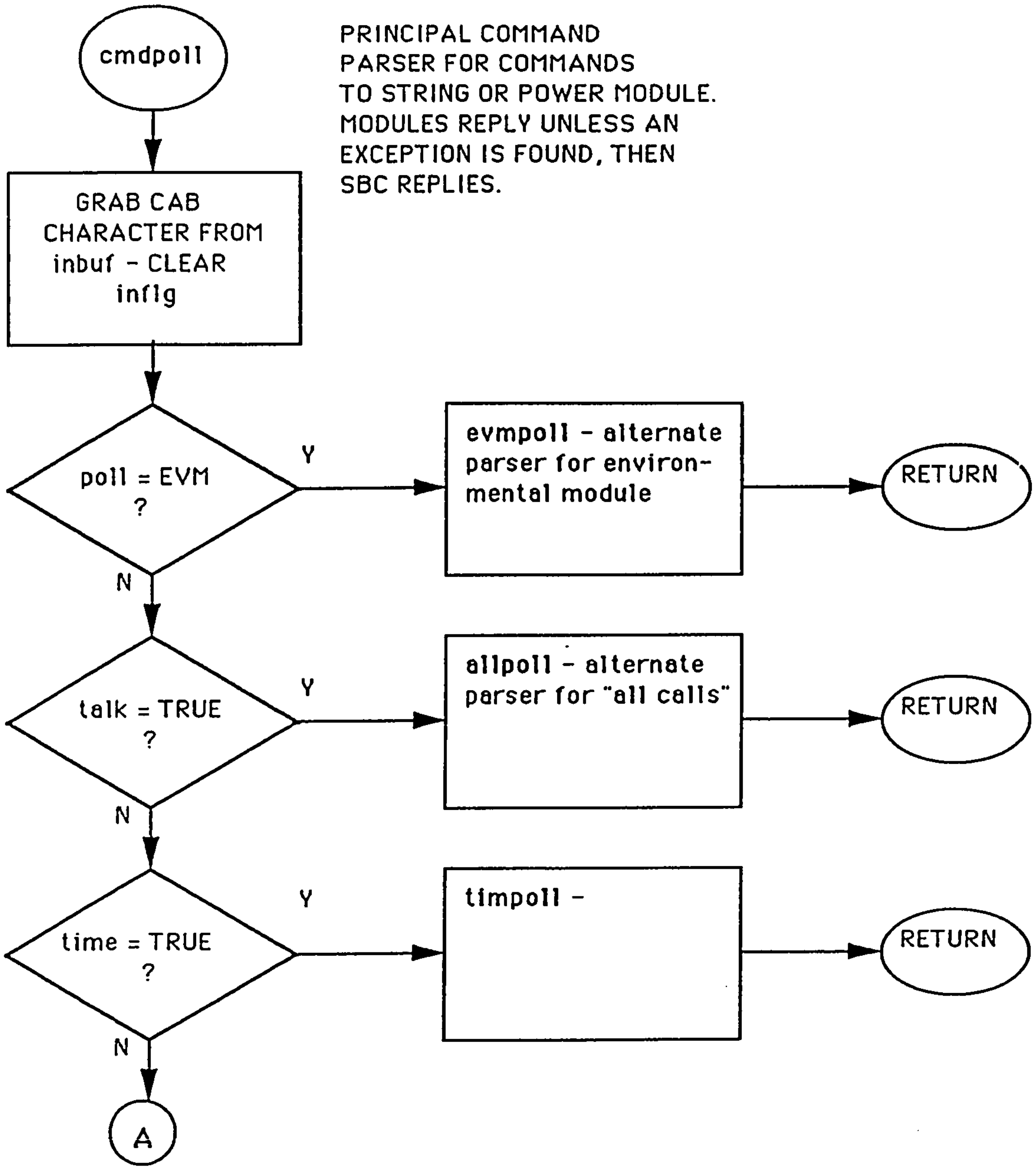

cmdpoll
PRINCIPAL COMMAND PARSER FOR COMMANDS TO STRING OR POWER MODULE. MODULES REPLY UNLESS AN EXCEPTION IS FOUND, THEN SBC REPLIES.
GRAB CAB CHARACTER FROM inbuf - CLEAR inflg
poll = EVM ?
Y
N
evmpoll - alternate parser for environ-mental module
RETURN
talk = TRUE ?
Y
N
allpoll - alternate parser for "all calls"
RETURN
time = TRUE ?
Y
N
timpoll -
RETURN
A

Figure 10.11 The main level routine cmdpoll

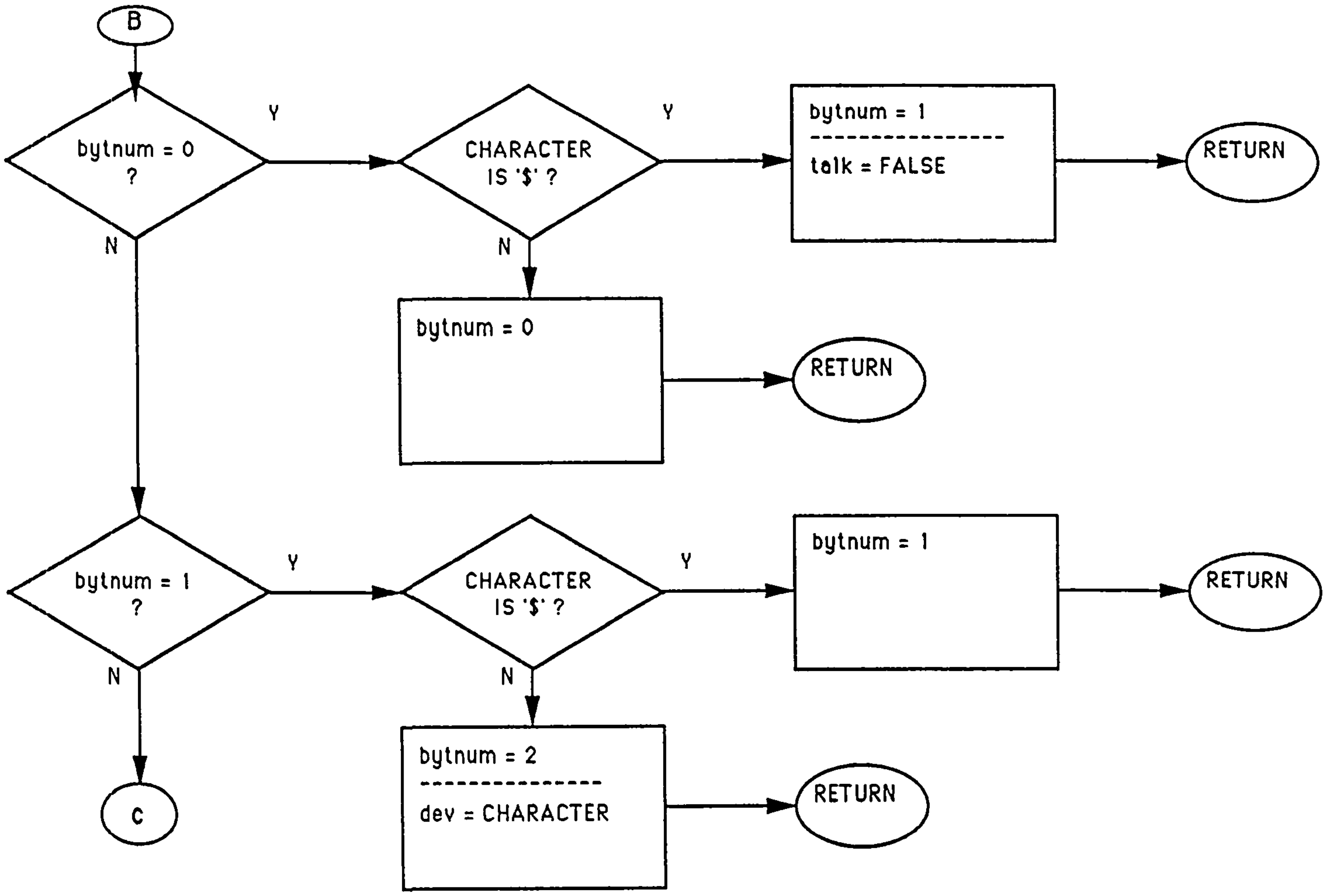


Figure 10.11 (Continued) The main level routine cmdpoll

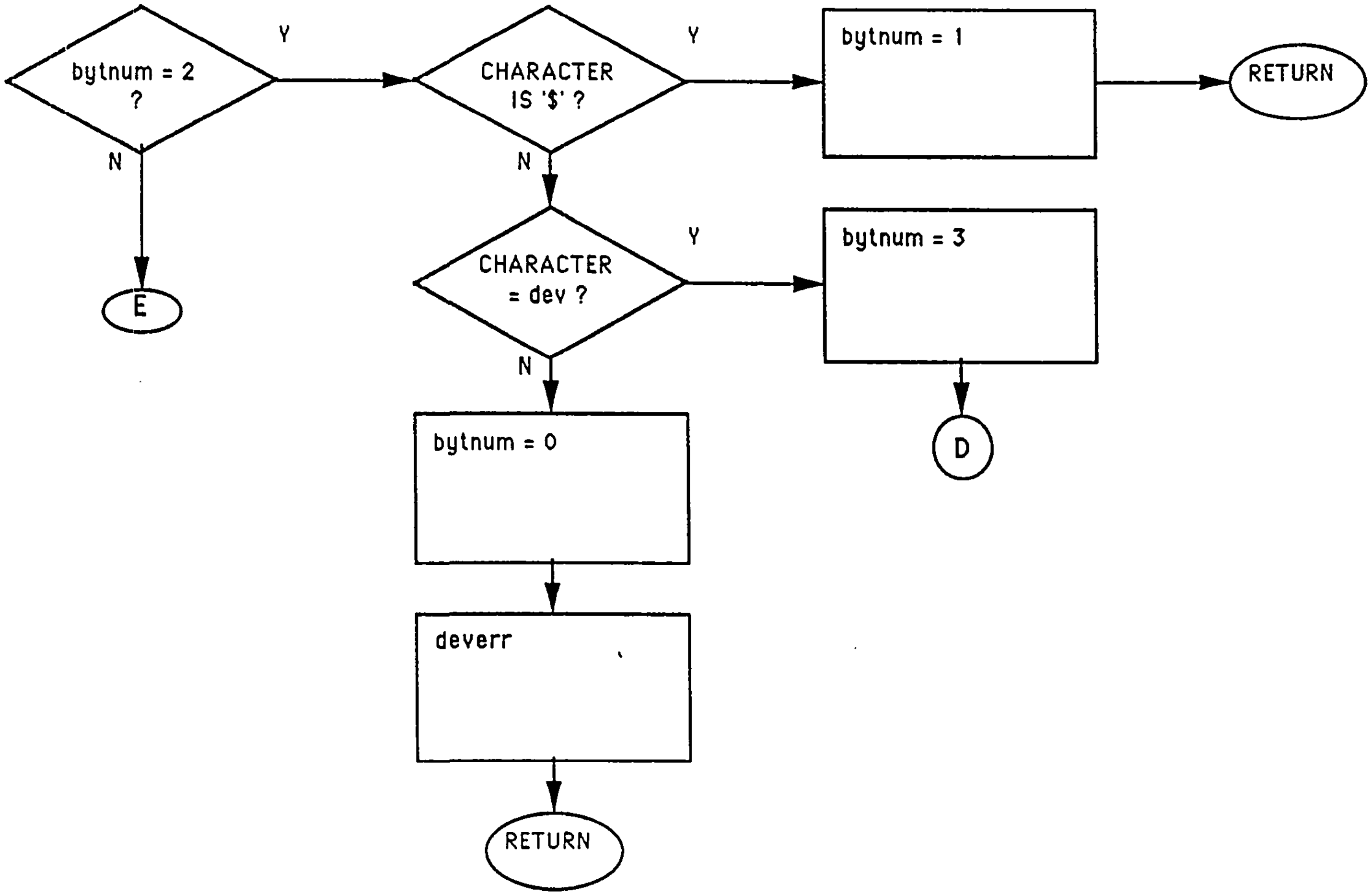


Figure 10.11 (Continued) The main level routine cmdpoll

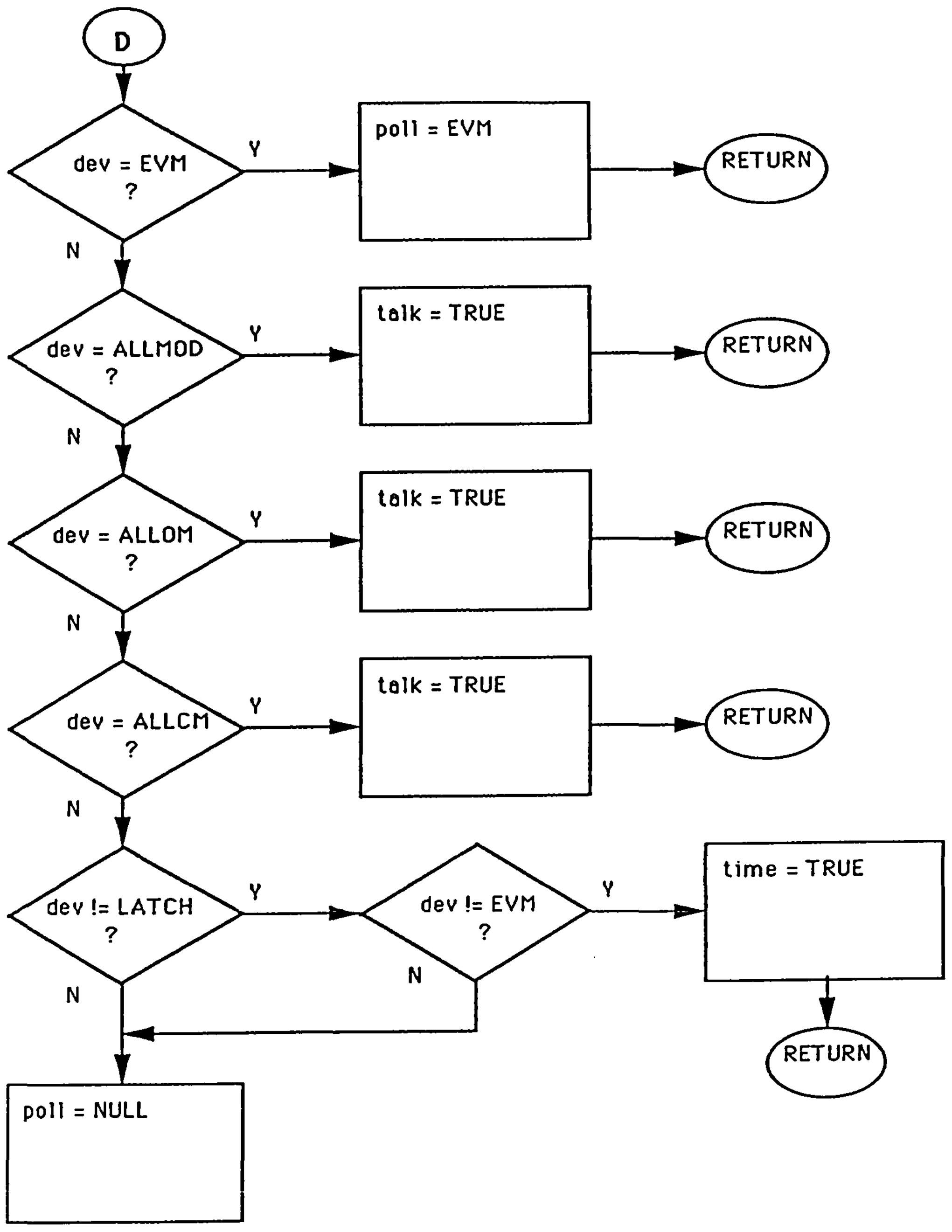


Figure 10.11 (Continued) The main level routine cmdpoll

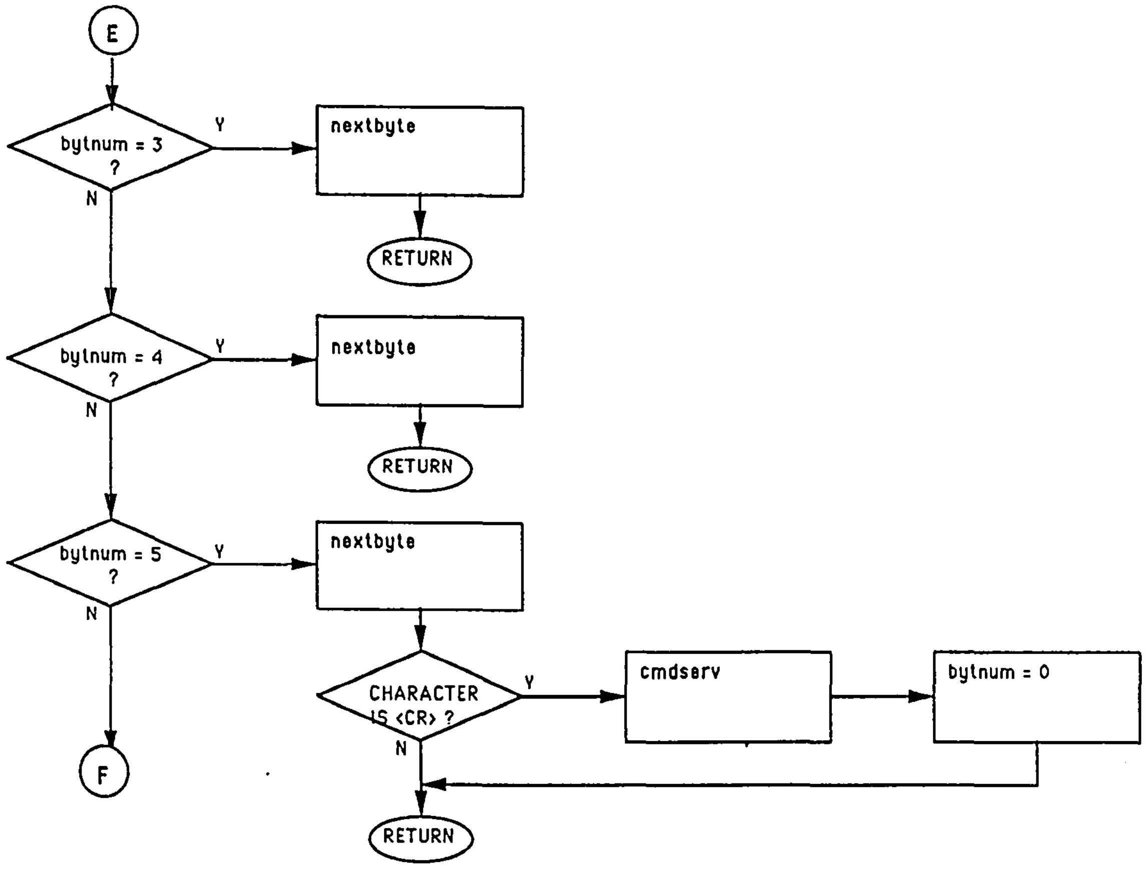


Figure 10.11 (Continued) The main level routine cmdpoll

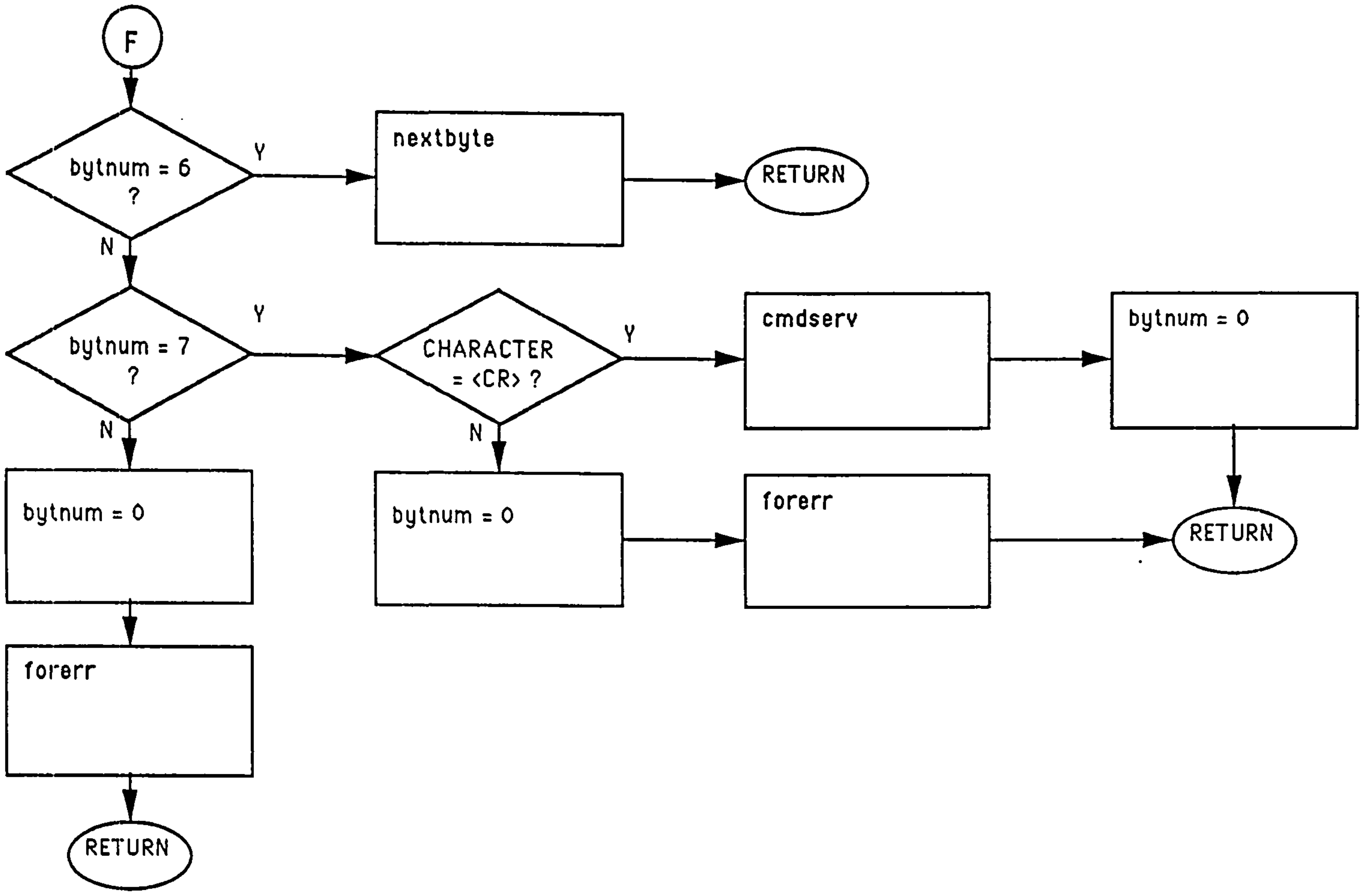


Figure 10.11 (Continued) The main level routine cmdpoll

Cmdserv is the last of the main level routine examples to be given here. It is called once the parsing logic decides that a complete command has been received and is in the command buffer. Cmdserv then either services the command or else calls an appropriate error routine before returning.

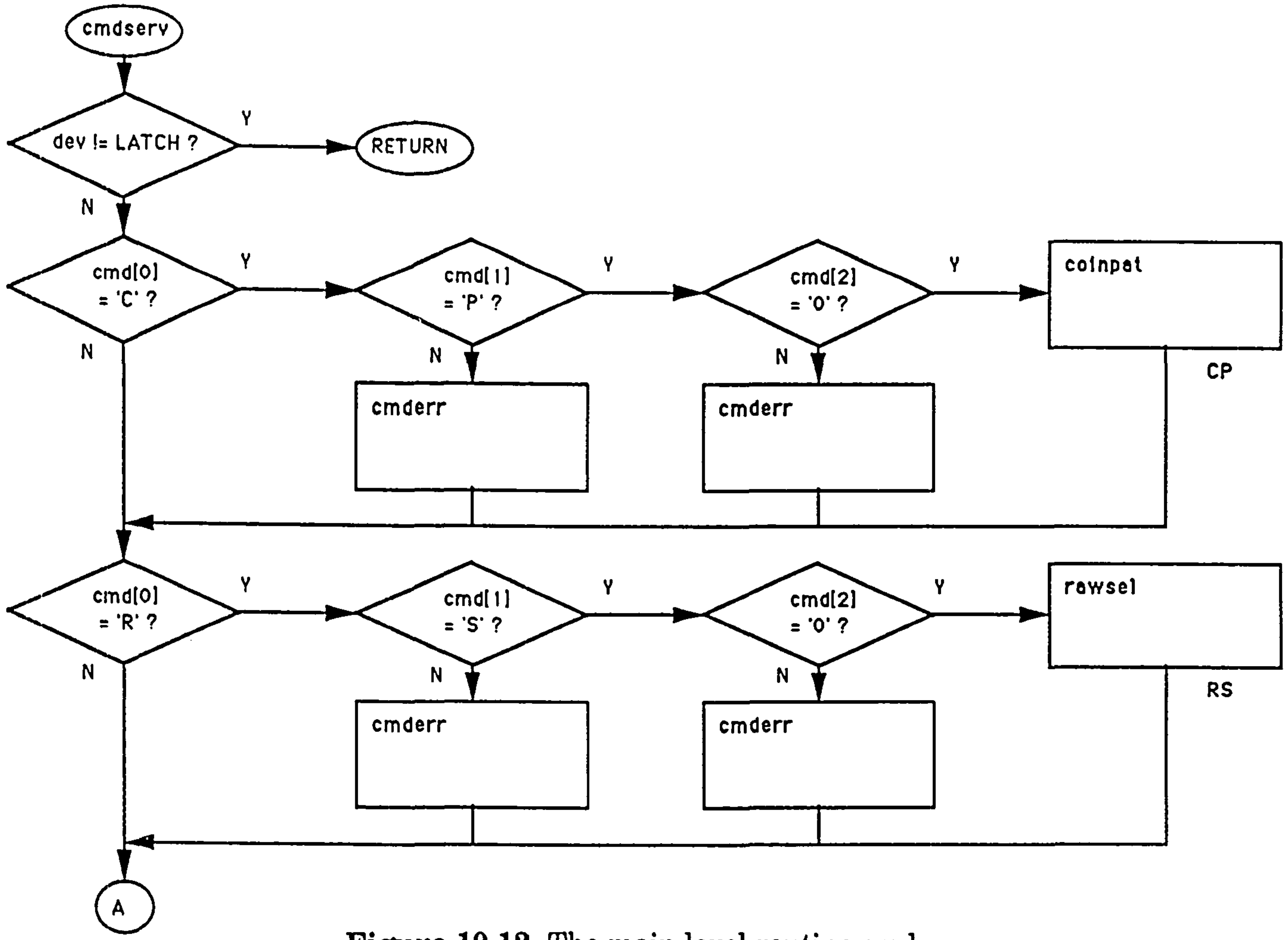


Figure 10.12 The main level routine cmdserv

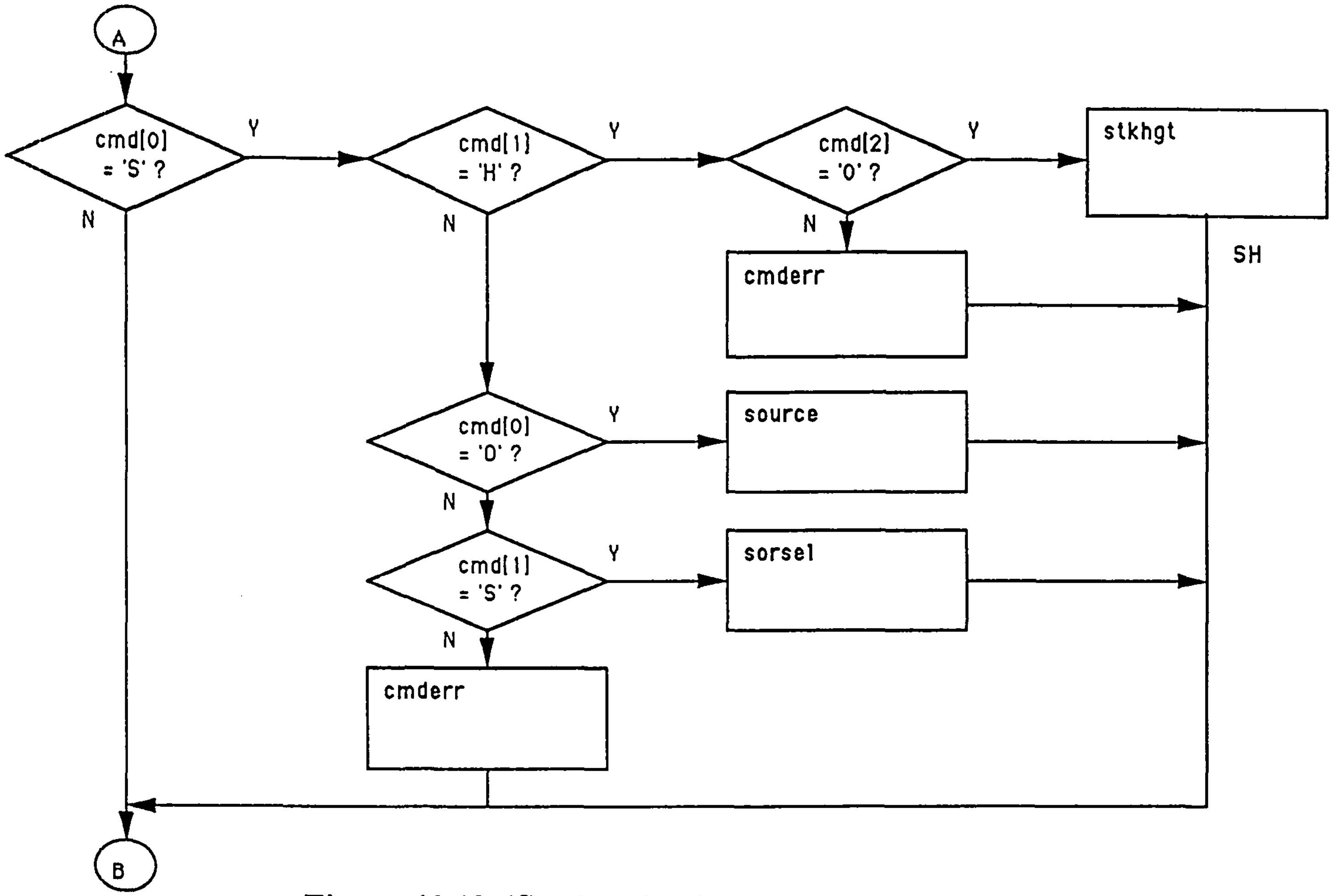


Figure 10.12 (Continued) The main level routine cmdserv

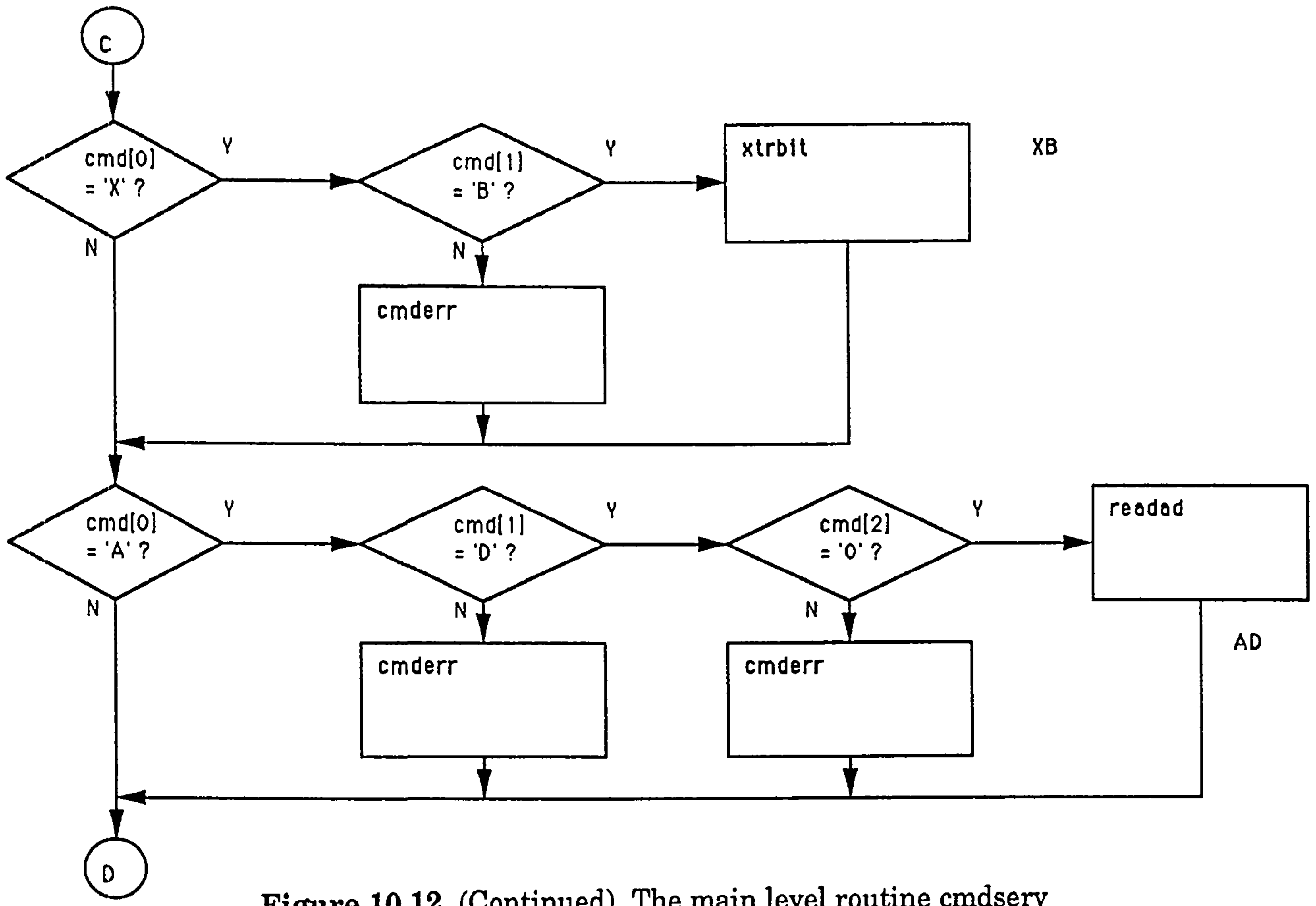


Figure 10.12 (Continued) The main level routine cmdserv

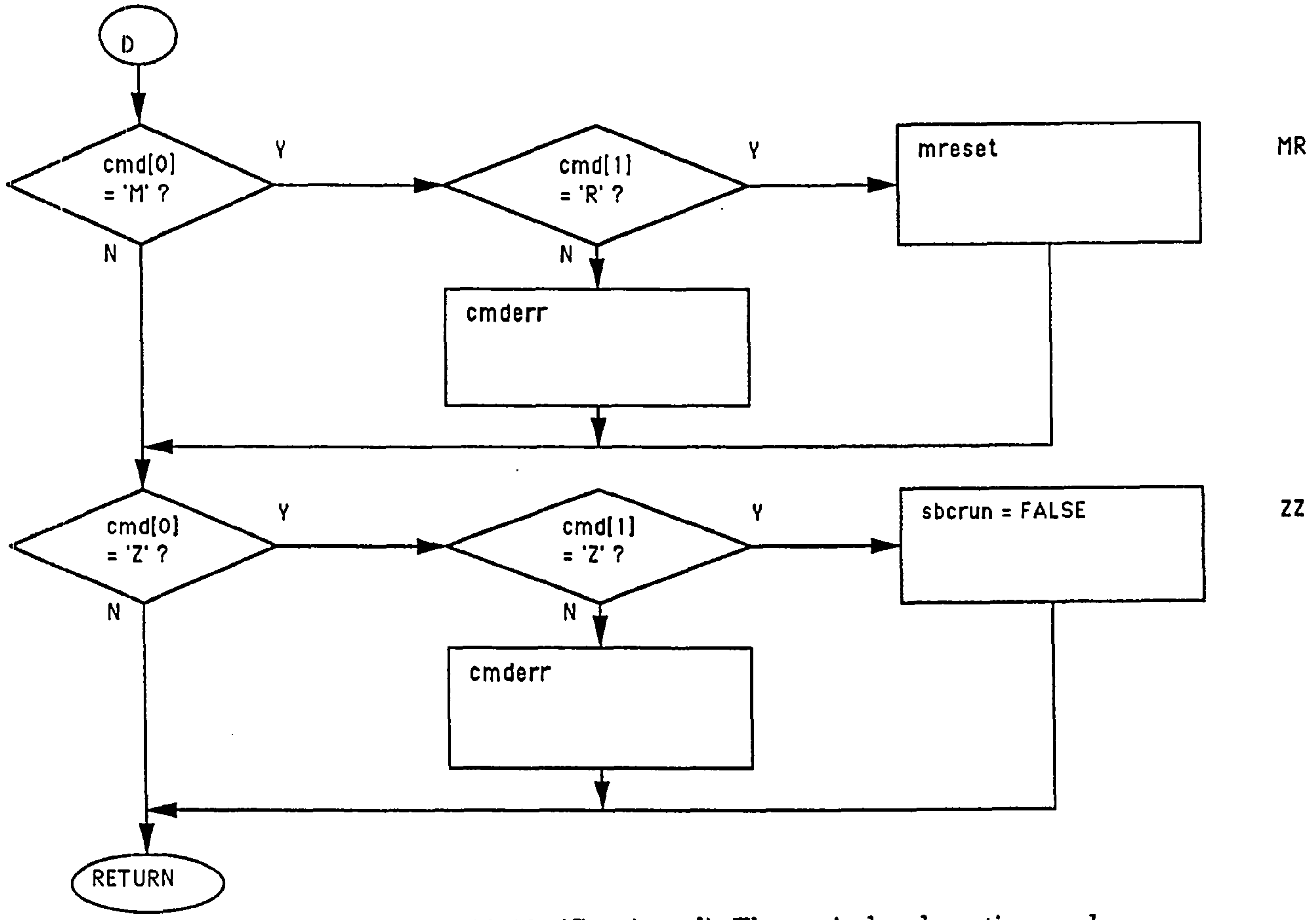


Figure 10.12 (Continued) The main level routine cmdserv

Readad is an example of a command level routine. Routines on this level set in motion the logic to execute a valid command. In particular, readad reads an analog (A to D) channel input returning the value read.

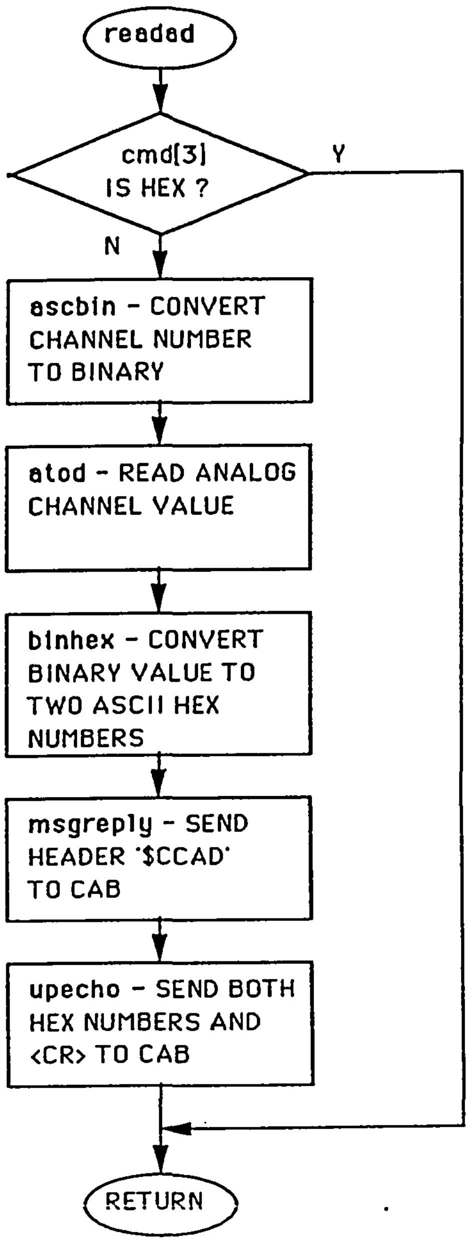

readad
cmd[3] IS HEX ?
Y
N
ascbin - CONVERT CHANNEL NUMBER TO BINARY
atod - READ ANALOG CHANNEL VALUE
binhex - CONVERT BINARY VALUE TO TWO ASCII HEX NUMBERS
msgreply - SEND HEADER '$CCAD' TO CAB
upecho - SEND BOTH HEX NUMBERS AND <CR> TO CAB
RETURN

Figure 10.13 The command level routine readad

Outport is an example of an I/O level routine. These routines direct information flow to or from the appropriate communications ports. In particular, outport directs a byte to a specified output port.

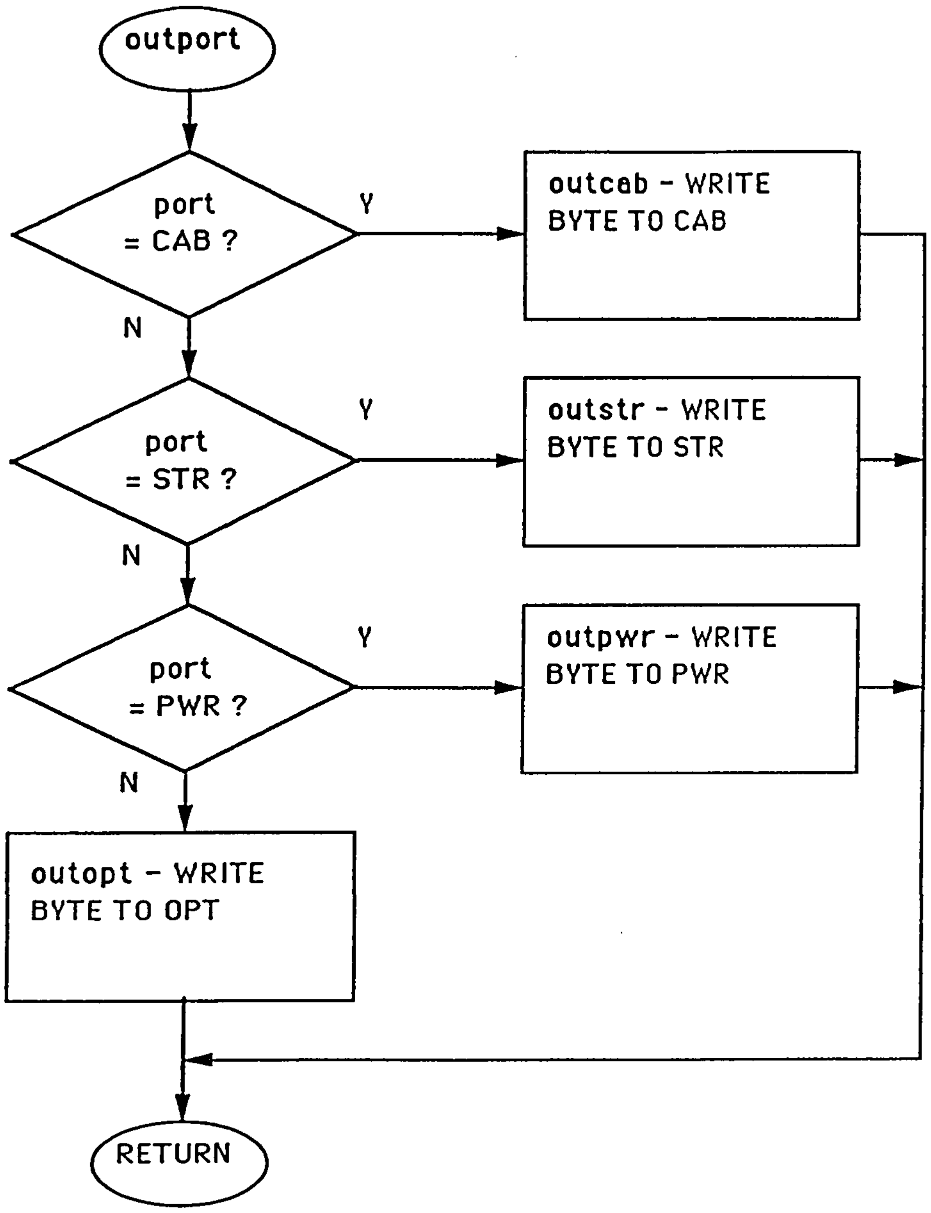

outport
port = CAB ?
Y
N
outcab - WRITE BYTE TO CAB
port = STR ?
Y
N
outstr - WRITE BYTE TO STR
port = PWR ?
Y
N
outpwr - WRITE BYTE TO PWR
outopt - WRITE BYTE TO OPT
RETURN

Figure 10.14 The I/O level routine outport

Cmderr is an example of an error level routines. Routines on this level generate error messages which are passed back up the cable through the ports CAB and OPT. In the case of cmderr, a message string

$??<DEV><DEV><CMD0><CMD1>...<CR>

where <DEV> is the one byte device identifier of either the SBC or the environmental module and <CMDn> are the successive command string bytes received.

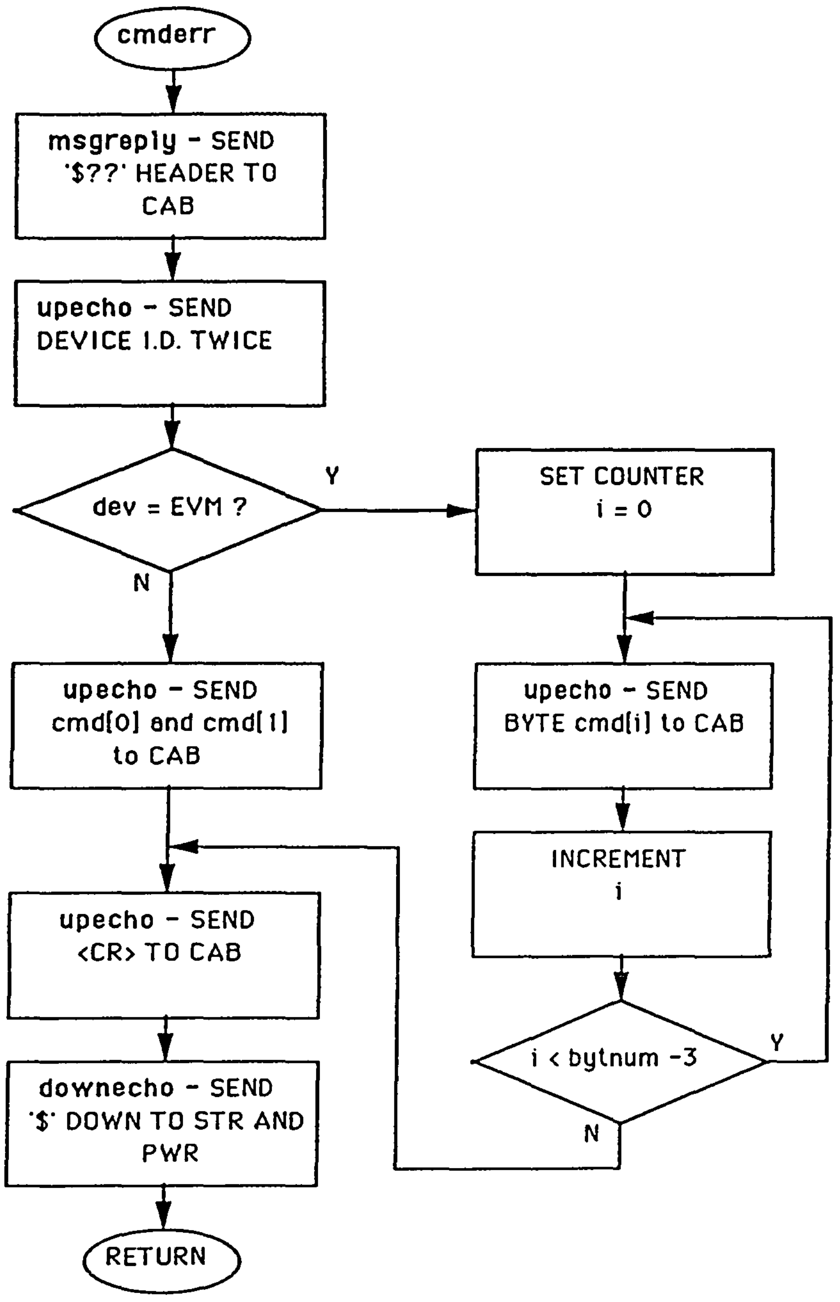

cmderr
msgreply - SEND '$??' HEADER TO CAB
upecho - SEND DEVICE I.D. TWICE
dev = EVM ?
Y
N
SET COUNTER i = 0
upecho - SEND cmd[0] and cmd[1] to CAB
upecho - SEND BYTE cmd[i] to CAB
INCREMENT i
upecho - SEND <CR> TO CAB
i < bytnum -3
Y
N
downecho - SEND '$' DOWN TO STR AND PWR
RETURN

Figure 10.15 The error level routine cmderr

Upecho is an example of a communications level routine. These routines logically direct information flow to multiple ports or generate the "fake" reply messages required in some of the exception coding. In particular, upecho directs a reply byte to both of the cable ports CAB and OPT for transmission back up to the ship board computer.

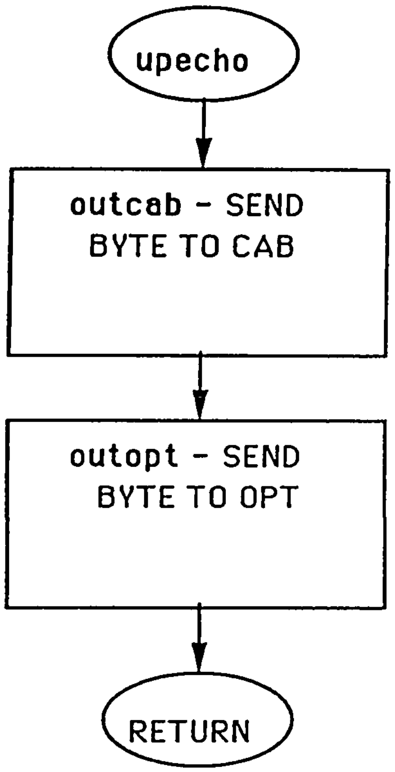


Figure 10.16 The communications level routine upecho

Ascbin is an example of a type checking and conversion level routine. Such routines either check to see if information presented to them is consistent with a particular representation or they convert information from one representation to another. In the case of ascbin, an ASCII hexadecimal byte is converted into its binary equivalent.

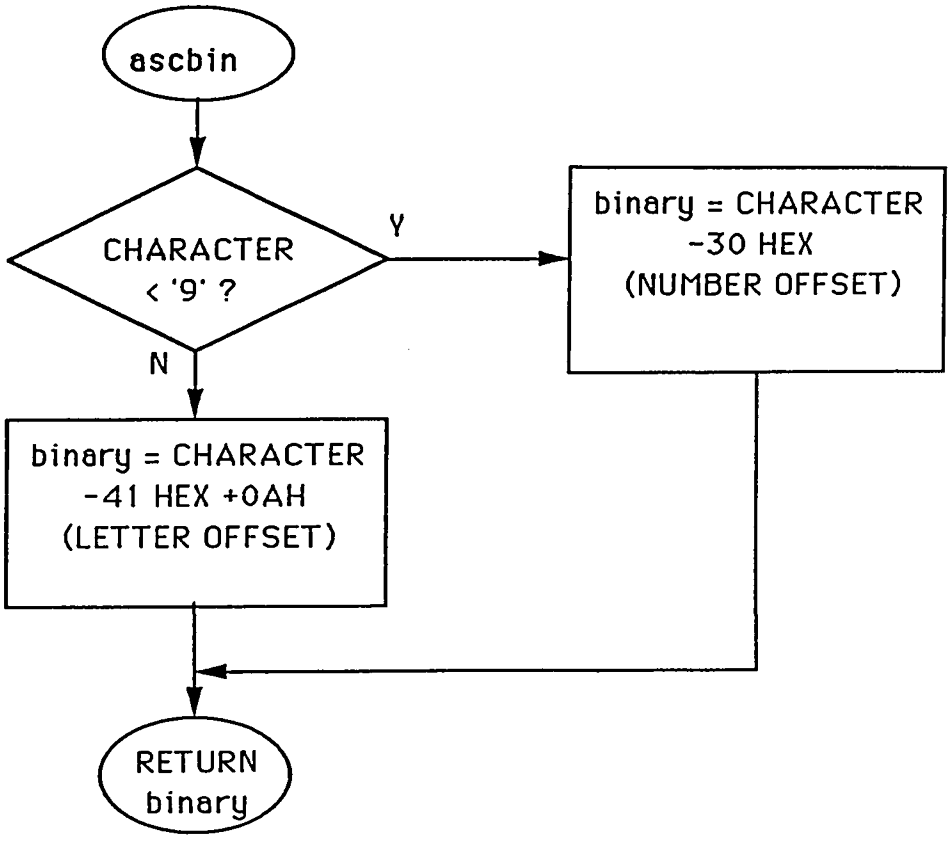

ascbin
CHARACTER < '9' ?
Y
N
binary = CHARACTER -30 HEX (NUMBER OFFSET)
binary = CHARACTER -41 HEX +0AH (LETTER OFFSET)
RETURN binary

Figure 10.17 The conversion level routine ascbin

Init is an example of an initialization and control routine. Basically the CAB and STR ports are set up for 300 baud communications.

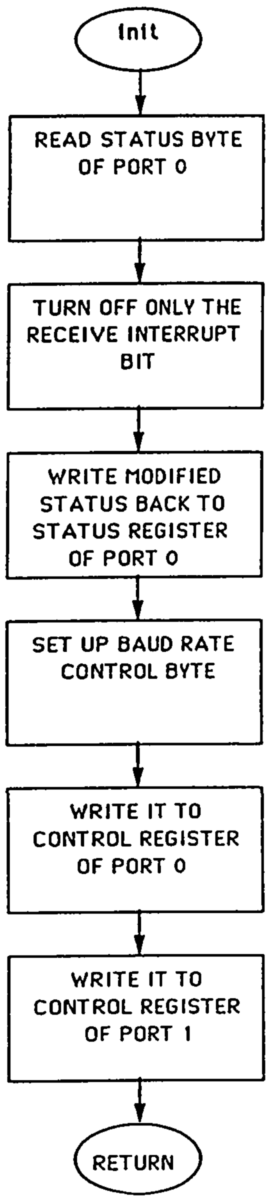

Init
READ STATUS BYTE OF PORT 0
TURN OFF ONLY THE RECEIVE INTERRUPT BIT
WRITE MODIFIED STATUS BACK TO STATUS REGISTER OF PORT 0
SET UP BAUD RATE CONTROL BYTE
WRITE IT TO CONTROL REGISTER OF PORT 0
WRITE IT TO CONTROL REGISTER OF PORT 1
RETURN

Figure 10.18 The initialization level routine init

Device drivers are those assembly language routines which directly drive the hardware. As such, most of the machine dependent code is to be found in the device drivers. Incab is an example of such a device driver. In particular, the function of incab is to fetch a byte from the port CAB.

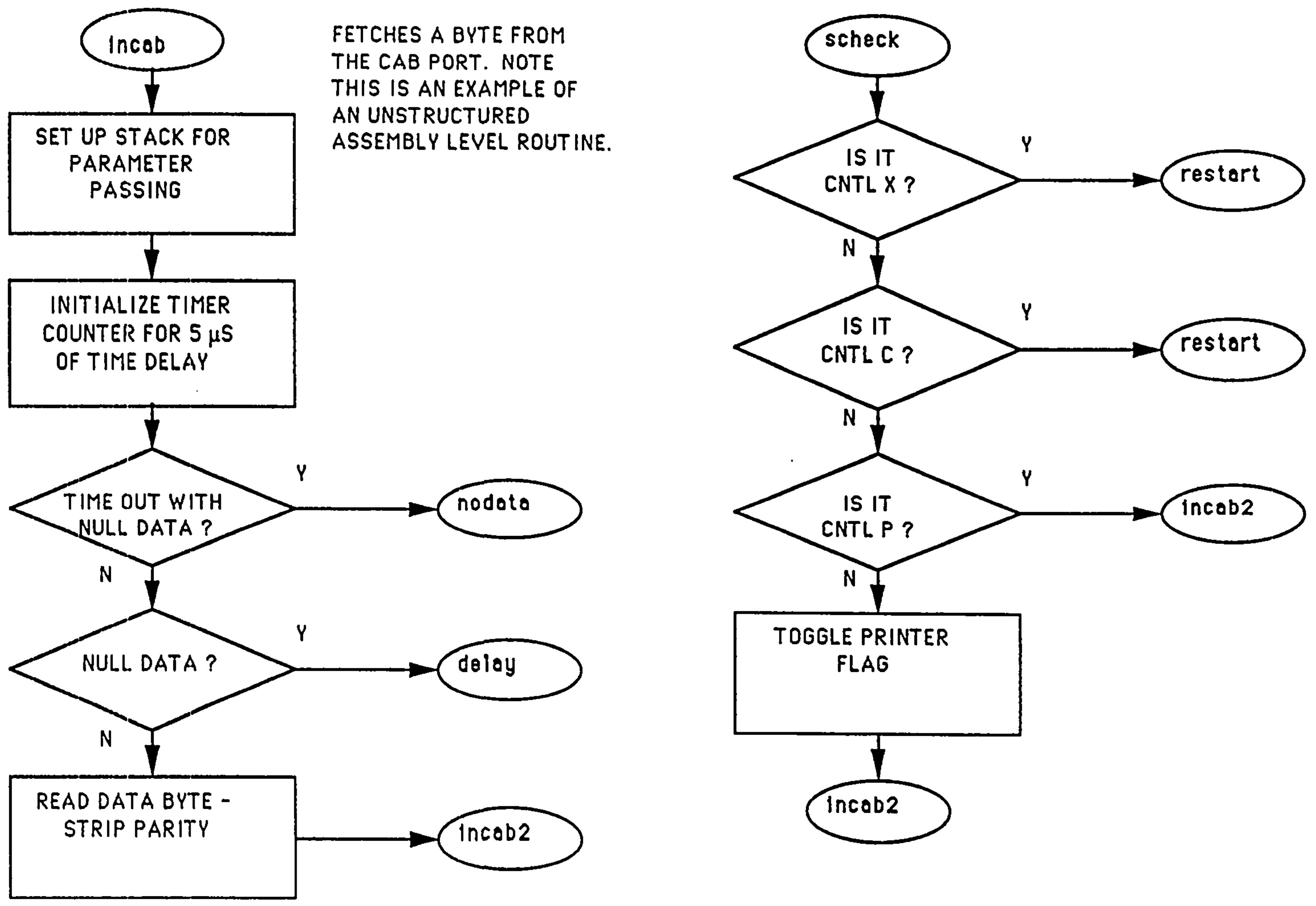

incab
FETCHES A BYTE FROM THE CAB PORT. NOTE THIS IS AN EXAMPLE OF AN UNSTRUCTURED ASSEMBLY LEVEL ROUTINE.
SET UP STACK FOR PARAMETER PASSING
INITIALIZE TIMER COUNTER FOR 5 μS OF TIME DELAY
TIME OUT WITH NULL DATA ?
Y
N
nodata
NULL DATA ?
Y
N
delay
READ DATA BYTE - STRIP PARITY
incab2
scheck
IS IT CNTL X ?
Y
N
restart
IS IT CNTL C ?
Y
N
restart
IS IT CNTL P ?
Y
N
incab2
TOGGLE PRINTER FLAG
incab2

Figure 10.19 The device driver routine incab

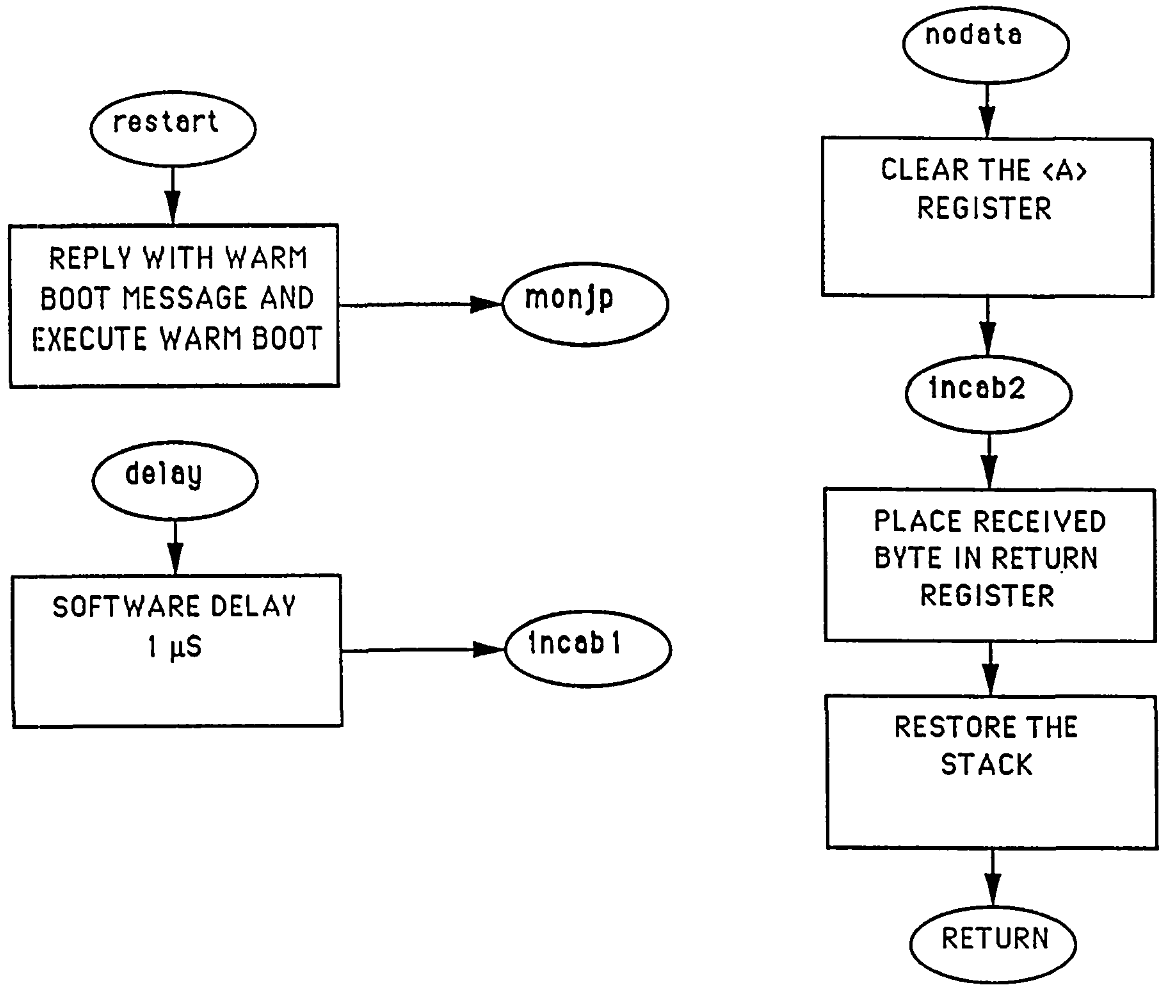


Figure 10.19 (Continued) The device driver routine incab

Chapter 11 – Analysis of the Short Prototype String Data

In this final chapter, some analysis of the data from the short prototype string is described. In particular, the data taken during the Fall 1987 cruise off the Kona coast of the island of Hawai'i is analyzed statistically to determine the cosmic ray muon background rate at a depth of 4 Km. Additionally, a determination of the power index of the cosine of the zenith angle is made for that depth.

11.1 A Purely Statistical Procedure for the Determination of the Cosmic Ray Muon Background Rate

A purely statistical procedure for determining the cosmic ray muon rate is possible by reading the scalar counters of the individual photomultiplier tube rates and then using them to compute what one would expect for an aggregate random coincidence rate. In turn, this is compared to the actual aggregate coincidence rate measured (and recorded in yet another scalar counter) and difference or "excess" rate taken as the cosmic ray muon rate. This contrasts with approaches to determining the cosmic ray muon rate based upon the fitting of individual events. For this procedure to work, several assumptions or conditions *must* be met. First, there can be no calibration pulses in the data. Second, all noise must be random, in other words there should be no bioluminescence forcing promotions, flashing photomultiplier tubes, etc. Thus, all events are either *random* (statistically determinable) or *real muons* (cosmic rays).

The aggregate probability P for m things each with a common finite individual probability of occurrence p tried n times is given by the binomial distribution

$$P(m) = \binom{n}{m} p^m q^{n-m} \quad \Big| \quad 0 < m \leq n \qquad (11.1a)$$

and

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}\,,\; q = 1 - p \tag{11.1b}$$

One could say that $\binom{n}{m}$ is a count (or weight) of the number of identical terms $p^m q^{n-m}$ making up the aggregate probability of an m-fold occurrence among n things.

Now, in the actual problem at hand, one is concerned with the random number of interchanges possible for $n = 7$ (or 6) optical modules with a coincidence triggering of $m = 5$ (or 4) and greater. The individual optical modules record different numbers of "hits" $h(s)$ in a given time interval ($t \sim 1$ s) yielding an individual rate of

$$r(s) = \frac{h(s)}{t} \;\Big|\; s = 1..7 \tag{11.2}$$

where s is a running index identifying the optical modules. In turn, these individual optical module rates result in individual probabilities

$$p(s) = r(s)\,\tau \tag{11.3}$$

where τ is the coincidence window used in the experiment (i.e. the time interval within which two or more optical modules *may* be considered to be causally connected in their recordings of a hit). Then, the task at hand is to predict what would be the expected *random* coincidence rate for n optical modules with a coincidence triggering of m or greater optical modules recording a hit in time τ.

Again, for a given n and m, one expects $\binom{n}{m}$ number of terms only the associated single probabilities $p(s)$ (s = 1..7) are not necessarily identical. Thus, the $\binom{n}{m}$ number of terms is the same as the number of possible interchanges in the term

$$p(1)\,p(2)\,p(3)\,...\,p(m)\,q(m+1)\,...\,q(n) \tag{11.4}$$

In other words,

$$\begin{aligned} P(m) = {} & p(1)\,p(2)\,p(3)\,...\,p(m)\,q(m+1)\,...\,q(n) \\ & + p(n)\,p(1)\,p(2)\,...\,p(m-1)\,q(m)\,...\,q(n-1) \\ & \qquad + ... + \\ & + p(2)\,p(3)\,p(4)\,...\,p(m+1)\,q(m+2)\,...\,q(1) \end{aligned} \tag{11.5}$$

For example, should one have n = 7 optical modules and be seeking the aggregate probability for $m \geq$ 5-fold coincidence then

$$P = P(7) + P(6) + P(5) \tag{11.6}$$

and the associated random rate within the coincidence window time τ is

$$r = \frac{P}{\tau} \tag{11.7}$$

The data was accumulated in a series of 24 bit wide (2^{24} = 16777216) scalars. Of importance to this calculation are the seven independent optical module scalars and the m-fold trigger scalars (most notably the 4-fold or greater and 5-fold or greater triggers). Approximately once every second, the data was read from the scalars and time stamped. Thus, the difference between two successive reads of a given scalar divided by the difference in elapsed time gives the associated rate. For this experiment, a coincidence is defined between two or more tubes if they register a hit within 160 ns of each other.

Two programs were developed to accomplish this data analysis. The first one, RATECMP, had the function of unpacking and reading the raw scalar data from the SPS data files. This data was read from the approximately one second interval stored data blocks and, with the actual computed elapsed time, the rates computed. Additional code for checking such things as counter overflow, etc. was introduced. Completely analogously, the counter recording aggregate 5-fold or greater trigger coincidences was also read and converted into a rate. In turn, all of these rates were written into a data file for future access. Figure 11.1 is a flow chart of this program while Table 11.1 shows the average rate for each of the photomultiplier tubes at four different depths.

Table 11.1 The individual average photomultiplier tube rates at different depths. Note that only six tubes were operating at a depth of 3.0 Km and that the only depth in which the rate sum did not saturate the instrument was at 4.0 Km.

Optical Module	Average Rate (10^4 Hz)			
Number	2.5 Km	3.0 Km	3.5 Km	4.0 Km
1	15.78	18.10	15.25	15.00
2	17.09	19.96	14.96	15.19
3	17.70	22.60	16.21	9.11
4	7.03	7.73	6.00	3.04
5	19.30	19.70	15.68	10.90
6	17.55	25.08	19.62	16.61
7	20.51	0.00	16.33	9.04
Sum	114.96	113.17	104.05	78.89

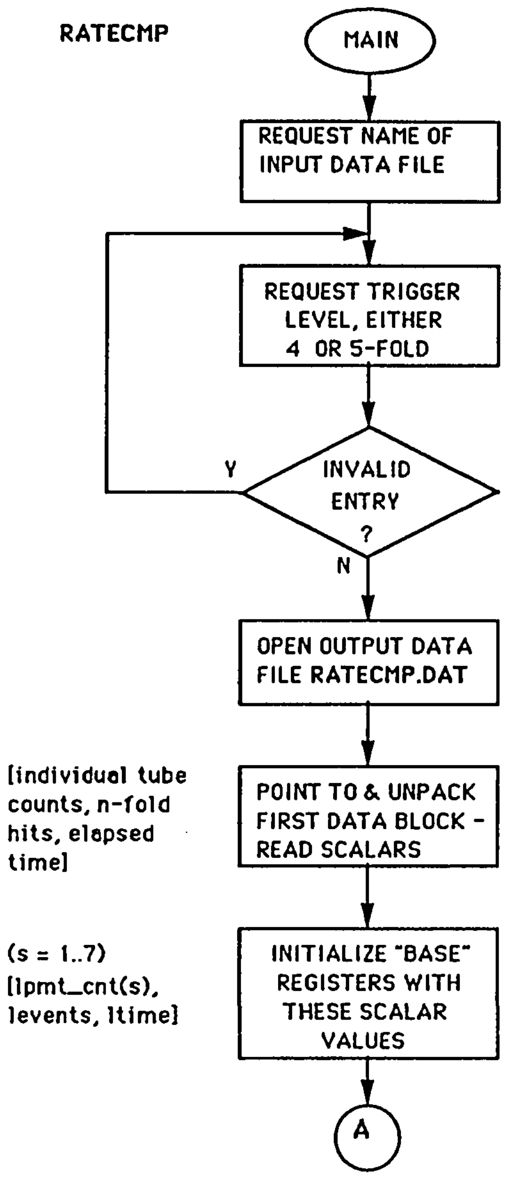

RATECMP
MAIN
REQUEST NAME OF INPUT DATA FILE
REQUEST TRIGGER LEVEL, EITHER 4 OR 5-FOLD
Y
INVALID ENTRY ?
N
OPEN OUTPUT DATA FILE RATECMP.DAT
[individual tube counts, n-fold hits, elapsed time]
POINT TO & UNPACK FIRST DATA BLOCK - READ SCALARS
(s = 1..7) [lpmt_cnt(s), levents, ltime]
INITIALIZE "BASE" REGISTERS WITH THESE SCALAR VALUES
A

Figure 11.1 The flow chart for the program RATECMP which returns to a file the individual optical module event rates as well as the measured aggregate 5-fold or greater event rate.

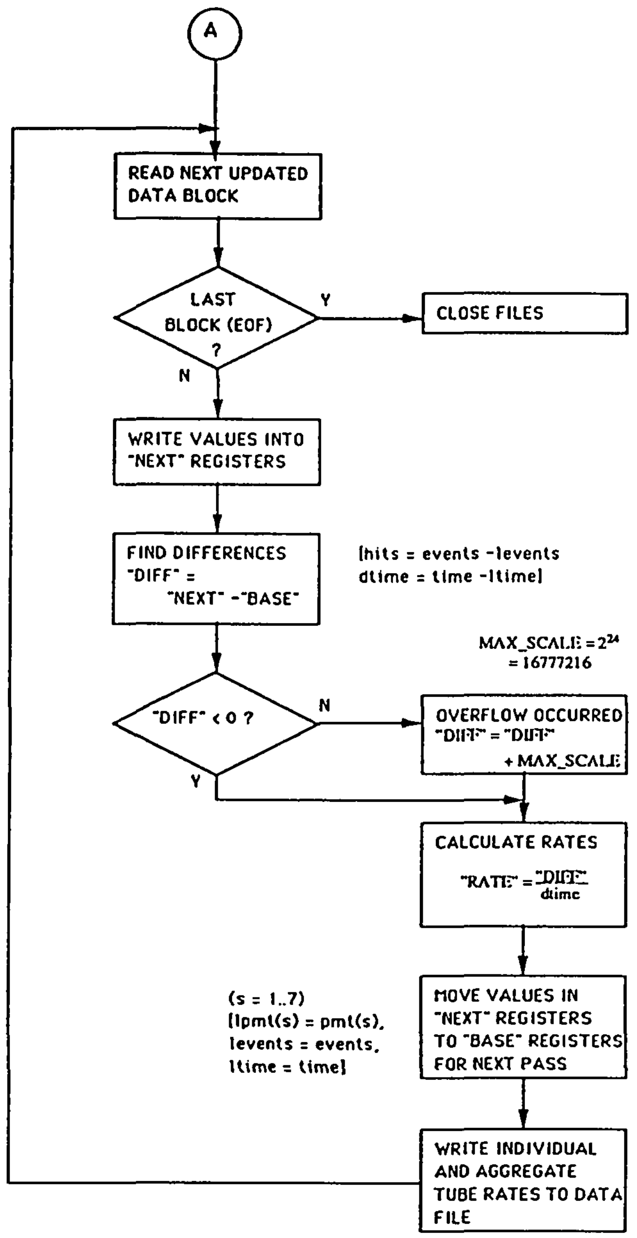
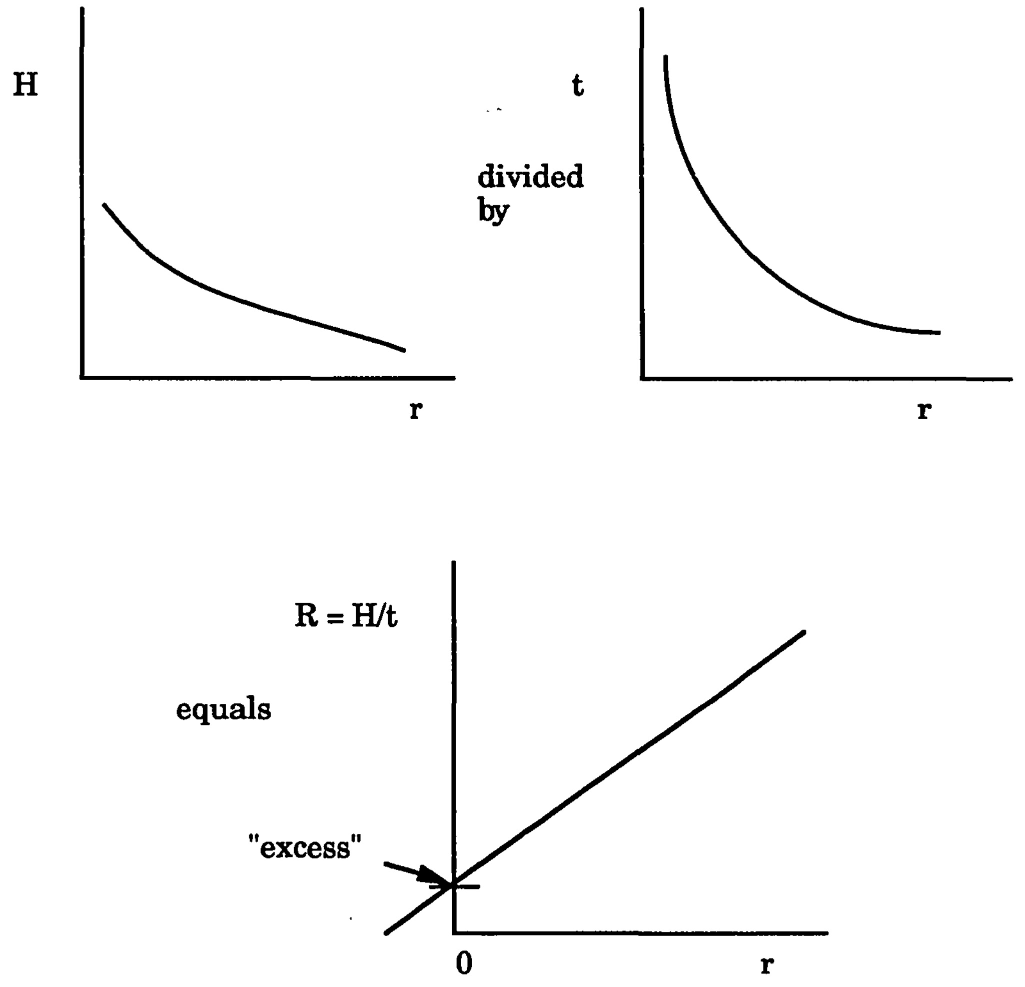

A
READ NEXT UPDATED DATA BLOCK
LAST BLOCK (EOF) ?
Y
N
CLOSE FILES
WRITE VALUES INTO "NEXT" REGISTERS
FIND DIFFERENCES "DIFF" = "NEXT" - "BASE"
[hits = events - levents dtime = time - ltime]
MAX_SCALE = 2^24 = 16777216
"DIFF" < 0 ?
N
Y
OVERFLOW OCCURRED "DIFF" = "DIFF" + MAX_SCALE
CALCULATE RATES "RATE" = "DIFF" / dtime
(s = 1..7) [lpmt(s) = pmt(s), levents = events, ltime = time]
MOVE VALUES IN "NEXT" REGISTERS TO "BASE" REGISTERS FOR NEXT PASS
WRITE INDIVIDUAL AND AGGREGATE TUBE RATES TO DATA FILE

Figure 11.1 (Continued) The flow chart for the program RATECMP which returns to a file the individual optical module event rates as well as the measured aggregate 5-fold or greater event rate.

The second program, RATEHST, calculates the expected random 5-fold or greater aggregate coincidence rate r and compares it to the actual 5-fold or greater aggregate coincidence rate R. First, the data file containing the rates of the optical modules and the 5-fold or greater coincidence hits is read on a block by block basis. Second, the expected 5, 6, and 7-fold random events summing on all possible interchanges of the tubes is calculated as contributions to a particular m-fold event. Third, the aggregate probability and thus a predicted rate for a given set of scalars is computed. Fourth, a series of histograms is plotted. The first one plots the number of actual events that are 5-fold or greater v.s. the predicted aggregate rate. The second one keeps track of the integration time t v.s. the predicted aggregate rate. Finally, the third one is formed by dividing the first (hits) histogram by the second (time) histogram to produce a scatter plot that reflects the aggregate observed rate v.s. predicted aggregate rate. These three plots are represented symbolically in Figure 11.2:



t = integration time [seconds]
H = number of hits (events)
r = predicted aggregate rate [Hz]
R = real aggregate rate [Hz]

Figure 11.2 A symbolic representation of the plots (two histograms and one scatter plot) used in finding the cosmic ray muon rate. The predicted aggregate rate *r* is the predicted aggregate probability *P* within the coincidence window τ.

The y-intercept of the "(predicted) rates vs. (observed) rates" histogram represents any excess 5-fold or greater coincidence count above and beyond what would be expected due to random coincidence of the optical modules for the actual individual rates encountered. Should the assumptions limiting contamination within the narrow 160 ns coincidence time window be upheld (such as no calibration pulses and minimum bioluminescence induced promotions), then the remaining excess counts should be entirely due to cosmic ray muons (or occasional cosmic ray neutrino induced muons). As such, the corresponding excess rate is the cosmic ray muon rate.

Figure 11.3 is the flow chart for RATEHST.

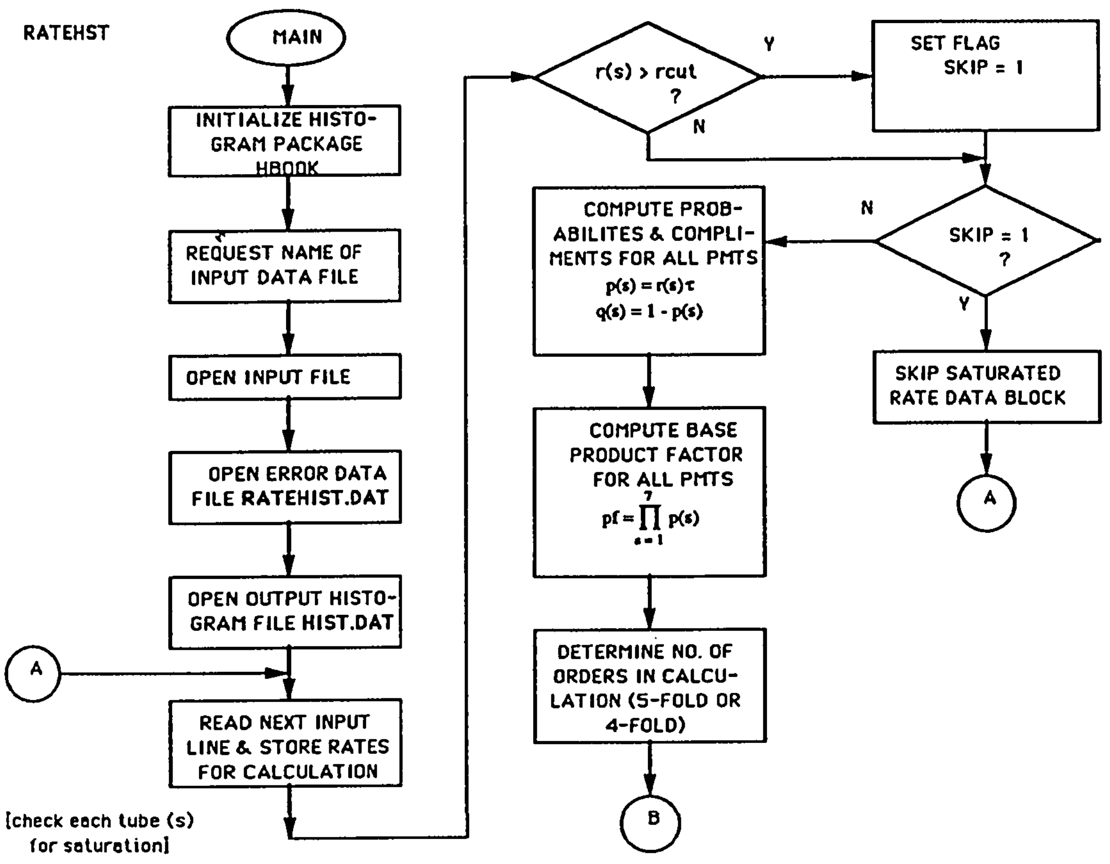

RATEHST
MAIN
INITIALIZE HISTOGRAM PACKAGE HBOOK
REQUEST NAME OF INPUT DATA FILE
OPEN INPUT FILE
OPEN ERROR DATA FILE RATEHIST.DAT
OPEN OUTPUT HISTOGRAM FILE HIST.DAT
A
READ NEXT INPUT LINE & STORE RATES FOR CALCULATION
[check each tube (s) for saturation]
r(s) > rcut ?
Y
N
SET FLAG SKIP = 1
SKIP = 1 ?
N
Y
COMPUTE PROBABILITES & COMPLIMENTS FOR ALL PMTS
p(s) = r(s)τ
q(s) = 1 - p(s)
COMPUTE BASE PRODUCT FACTOR FOR ALL PMTS
pf = ∏_{s=1}^{7} p(s)
DETERMINE NO. OF ORDERS IN CALCULATION (5-FOLD OR 4-FOLD)
B
SKIP SATURATED RATE DATA BLOCK
A

Figure 11.3 The flow chart for the program RATEHST which reads a data file of individual optical module event rates and then computes the expected aggregate 5-fold or greater coincidence random event rate r and compares it to the measured aggregate 5-fold or greater event rate R to extract the true cosmic ray muon background rate.

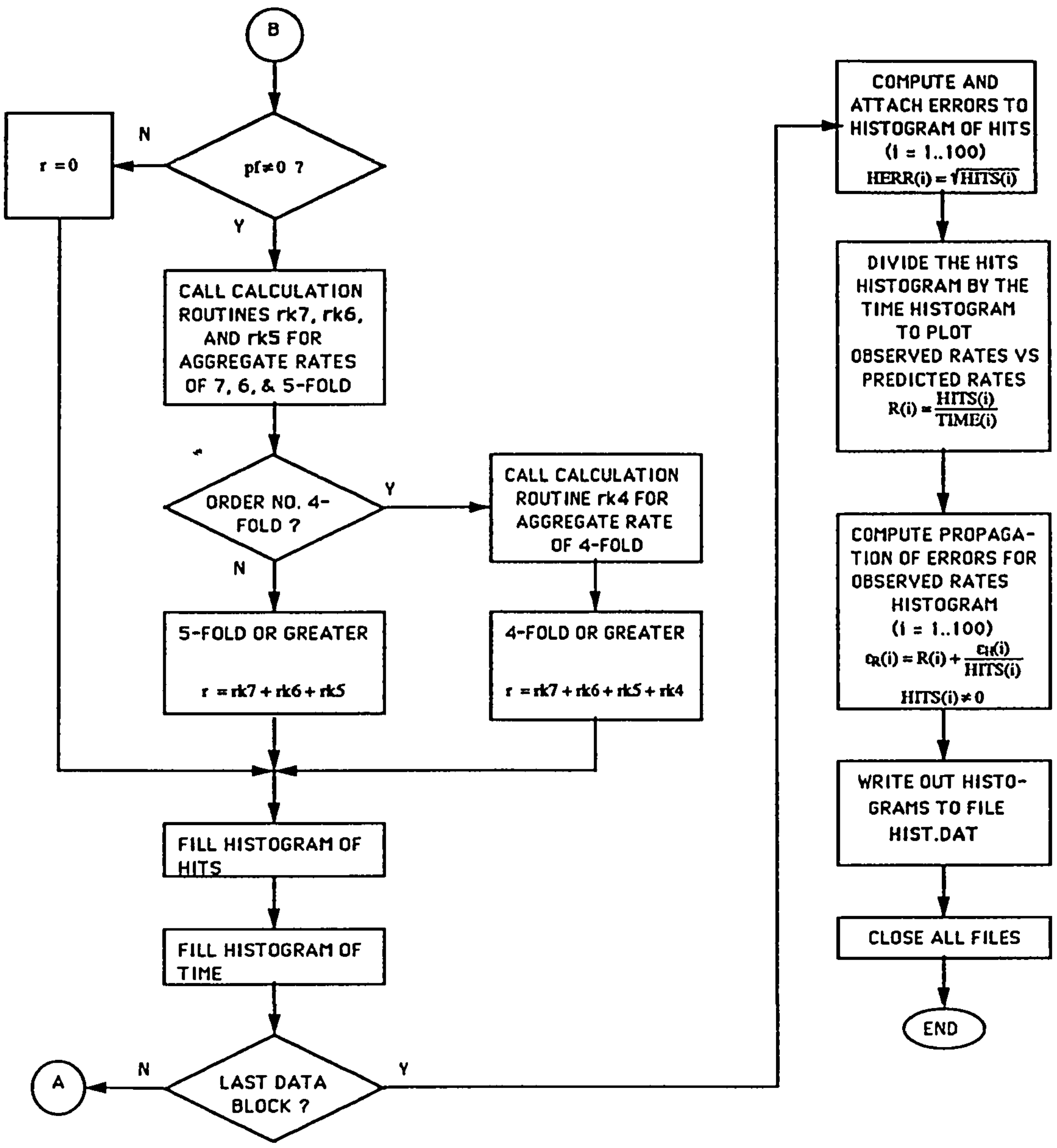

B
pf≠0 ?
N
Y
r = 0
CALL CALCULATION ROUTINES rk7, rk6, AND rk5 FOR AGGREGATE RATES OF 7, 6, & 5-FOLD
ORDER NO. 4-FOLD ?
Y
N
CALL CALCULATION ROUTINE rk4 FOR AGGREGATE RATE OF 4-FOLD
5-FOLD OR GREATER
r = rk7 + rk6 + rk5
4-FOLD OR GREATER
r = rk7 + rk6 + rk5 + rk4
FILL HISTOGRAM OF HITS
FILL HISTOGRAM OF TIME
LAST DATA BLOCK ?
N
Y
A
COMPUTE AND ATTACH ERRORS TO HISTOGRAM OF HITS (I = 1..100)
HERR(i) = √HITS(i)
DIVIDE THE HITS HISTOGRAM BY THE TIME HISTOGRAM TO PLOT OBSERVED RATES VS PREDICTED RATES
R(i) = HITS(i)/TIME(i)
COMPUTE PROPAGATION OF ERRORS FOR OBSERVED RATES HISTOGRAM (I = 1..100)
σR(i) = R(i) + σH(i)/HITS(i)
HITS(i) ≠ 0
WRITE OUT HISTOGRAMS TO FILE HIST.DAT
CLOSE ALL FILES
END

Figure 11.3 (Continued) The flow chart for the program RATEHST which reads a data file of individual optical module event rates and then computes the expected aggregate 5-fold or greater coincidence random event rate r and compares it to the measured aggregate 5-fold or greater event rate R to extract the true cosmic ray muon background rate.

The errors associated with this data analysis are generated as follows. To begin with, times are known to essentially arbitrary precision. However, the errors associated with the count of hits for the individual optical modules would be statistical. Now, since most of the time, an optical module does not see a flash, one has a sparse number of events compared to the total possible number of events. As such, this is the condition for a Poisson distribution (the binomial distribution for the case of $n \rightarrow \infty, p \rightarrow 0$) where the error ε_i is related to the count H_i in a given bin i by

$$\varepsilon(H_i) = \sqrt{H_i} \tag{11.8}$$

In the final plot, namely where the predicted rates are compared to the actual rates, the propagation of errors is simply the scaling

$$\varepsilon(R_i) = \varepsilon(H_i)\frac{R_i}{H_i} \tag{11.9}$$

Finally, a least squares straight line fit is made to the scatter plot and the reported fit error to the error bars on the data is taken as the error in determining the cosmic ray muon background rate.

Recall that photomultiplier tubes are intrinsically good random noise sources. As such, to detect something, one needs to add a threshold control to cut back the rate of random discharges. This cuts back but does not eliminate the random rate. The gain is then adjusted to allow detection of a light flash that exceeds the gain setting. Thus, one has the picture of relatively frequent random noise with occasional actual hits detected. Finally, to distinguish between random photomultiplier shot noise and an actual detection, photomultipliers are run in coincidence.

Now, the predicted aggregate rates of an m-fold coincidence among a collection of photomultiplier tubes was shown above to be dependent upon the aggregate probability of the coincidence occurring. This aggregate probability is the sum of the contributing probabilities of the m-fold or greater coincidence. In turn, these contributing probabilities are dependent upon the individual probabilities of the photomultiplier tubes which are derived from the individual rates of the photomultiplier tubes. As such, if a photomultiplier tube is exposed to an essentially time averaged constant level of light at a fixed voltage (gain) and threshold setting, then p is taken to be essentially constant for a given photomultiplier tube .

$$p = \text{constant} \tag{11.10}$$

It follows dimensionally that if one were to sample the time t spent by a photomultiplier tube at a rate r

$$t = \frac{p}{r} = \frac{1}{r}\text{constant} \tag{11.11}$$

With such a reciprocal relationship, one should obtain a plot similar to figure 11.4:

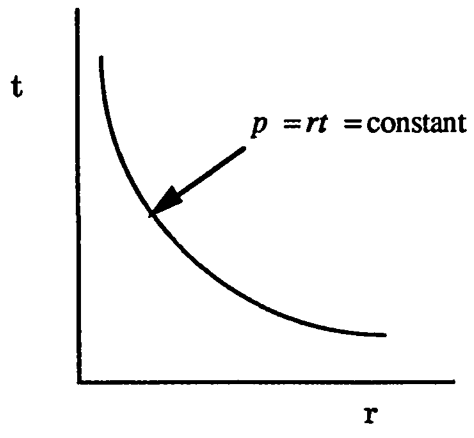


Figure 11.4 The symbolic plot of the reciprocal function of the time vs. rate of a single photomultiplier tube at a fixed gain and threshold exposed to a time averaged constant light source.

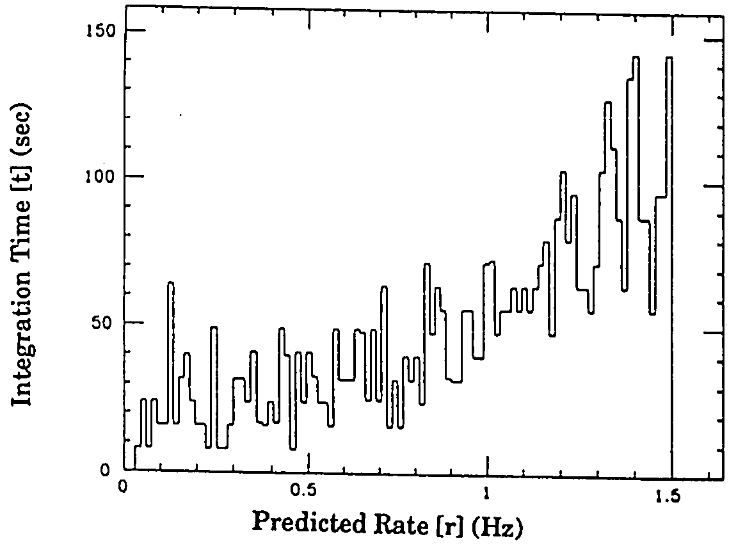
It follows, that since any aggregate probability P is a sum of products of the individual probabilities it is the product and sum of constants which in turn must be a constant so that the above dimensional argument also holds for the aggregate probability namely

$$P = rt = \text{constant} \tag{11.12}$$

and the form of the plot ought to be identical to that of figure 11.4 above.

From the above argument, it is easiest to describe first the histogram of the integration times t collected in bins of predicted aggregate ($m \geq 5$) random coincidence rate r. Figure 11.5 compares the time plots at different depths.

Predicted Rate vs. Integration Time at 2.5 km



Predicted Rate vs. Integration Time at 3.0 km

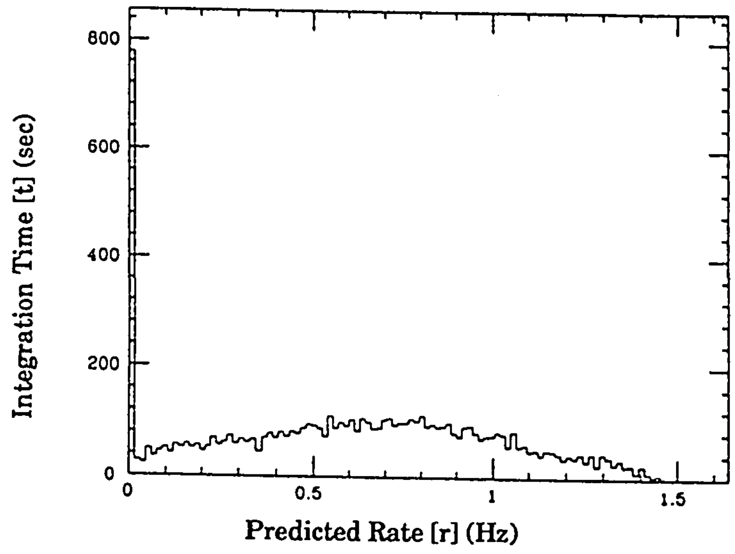
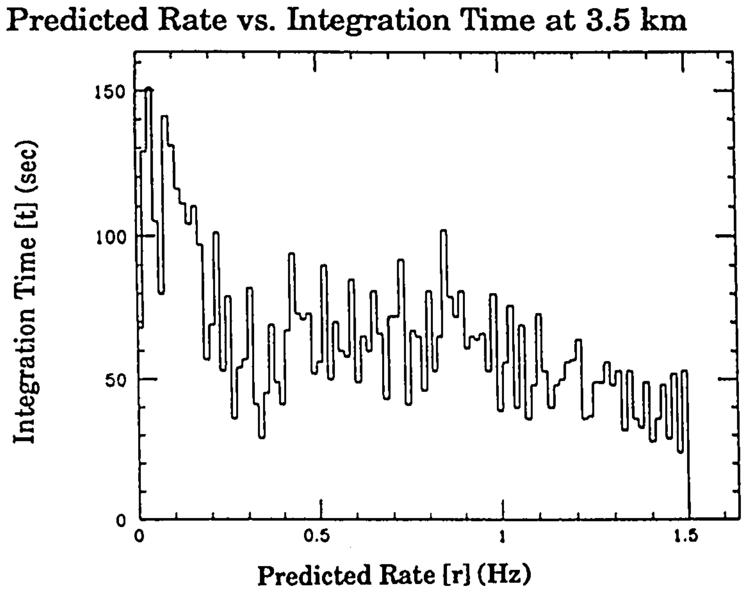


Figure 11.5a The (integration) time vs. predicted rates histograms computed for the first two of the four depths, namely at 2.5 Km and 3.0 Km respectively.



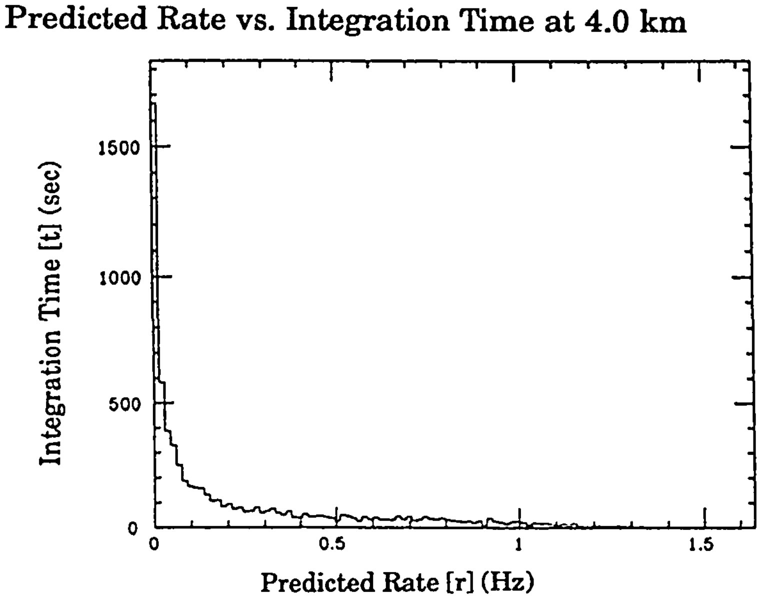
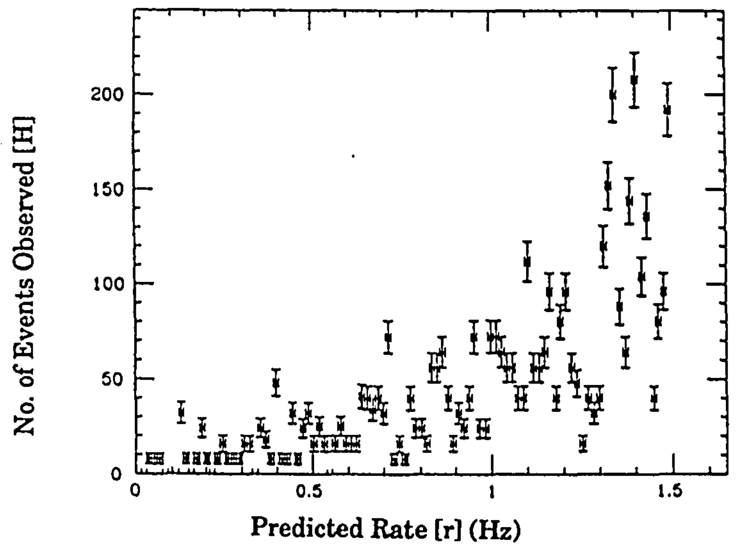


Figure 11.5b The (integration) time vs. predicted rates histograms computed for the last two of the four depths, namely at 3.5 Km and 4.0 Km respectively.

Comparing the four integration time plots, a pattern of sorts emerges. Starting at the greatest depth, namely 4.0 Km, one finds a histogram whose limiting distribution can approximated by an exponential decay or reciprocal function, i.e. exactly the form predicted above. At 3.5 Km, the general decay distribution now has a peak of excess predicted rates superimposed on it. This pattern continues through 3.0 Km to the 2.5 Km case where the now excess peak completely dominates at shallower depths. From this, one concludes that only the 4.0 Km depth is producing data that is totally random. In some sense, the other depths are contaminated with non-random sources.

Figure 11.6 compares the observed events (hits) plots at different depths. The comparative pattern is similar to that observed in the time plots with an excess peak of events already prominent in the 3.5 Km plot compared to the 4.0 Km plot and completely dominating in the 2.5 Km and 2.0 Km plots. The 4.0 Km plot is generally sensible in that it says that the number of hits occurring occasionally (i.e. at a low predicted aggregate rate *r*) is greater than the number of hits occurring more often (i.e. at a high predicted aggregate rate *r*).

Predicted Rate vs. Events Observed at 2.5 km



Predicted Rate vs. Events Observed at 3.0 km

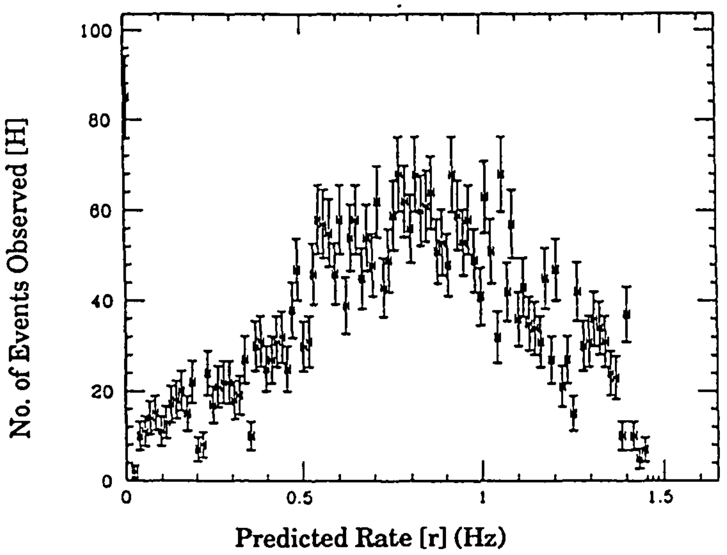
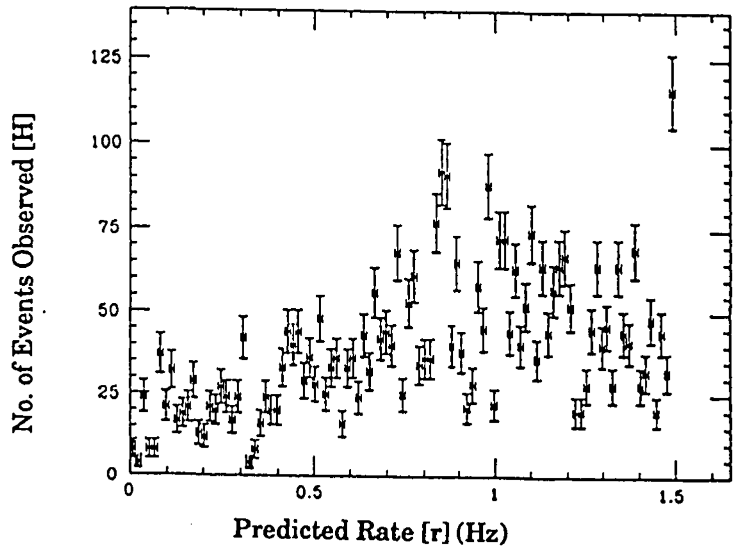


Figure 11.6a The events vs. predicted rates histograms computed for the first two of the four depths namely 2.5 Km and 3.0 Km, respectively.

Predicted Rate vs. Events Observed at 3.5 km



Predicted Rate vs. Events Observed at 4.0 km

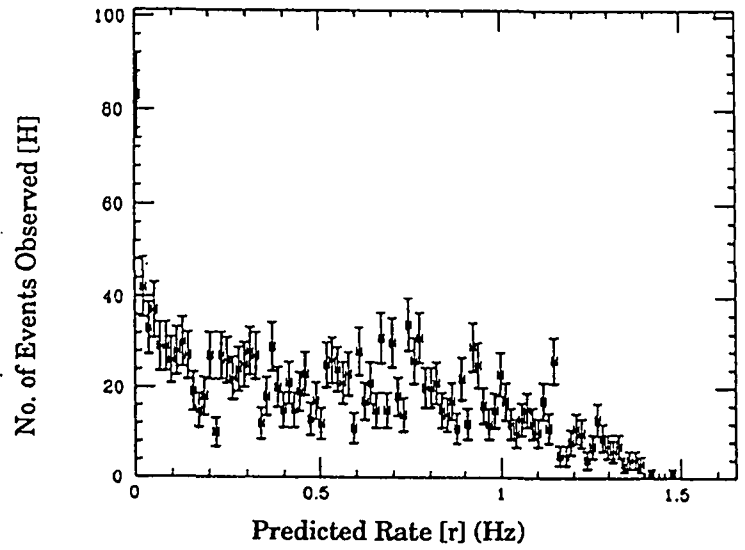


Figure 11.6b The events vs. predicted rates histograms computed for the last two of the four depths namely 3.5 Km and 4.0 Km, respectively.

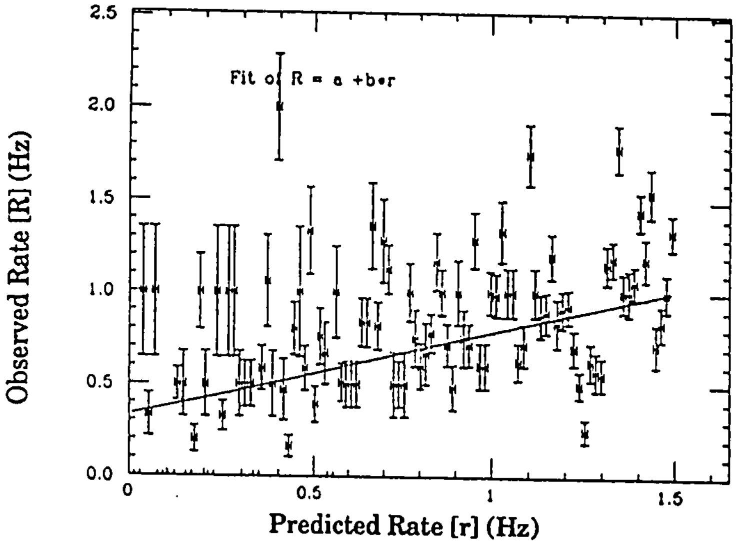
Finally, Figure 11.7 plots the observed rate R vs. the predicted rate r. Now, if all the observed rates were due to random coincidence only within the 160 ns window then the observed and random rates should be identical and a straight line plot going through the origin would be expected. A non-random source, namely a "real" coincidence in the form of a cosmic ray muon would manifest itself as an "observed" excess and thus shift the y-intercept above the ordinate. Roughly speaking, the 4.0 Km, 3.5 Km, and 3.0 Km plots exhibit this linear relationship while the 2.5 Km plot only in the most crude fashion does. The straight lines superimposed upon the data are least squares fits to the data. In the case of the 3.0 Km and 3.5 Km plots, the χ^2 confirm good fits are not possible as is obvious from the error bars. The χ^2 are summarized in Table 11.2:

Table 11.2 - the χ^2 per degree of freedom of the linear fits of the predicted rates vs. actual rates as a function of depth

DEPTH (Km.)	χ^2	Degrees of Freedom	χ^2 / DofF	Confidence Level of Fit
2.5	582.3	89	6.5	0.0000
3.0	140.8	95	1.5	0.0016
3.5	403.4	98	4.1	0.0000
4.0	91.50	93	0.98	0.5246

Thus, altogether, one is lead to the conclusion that the only depth at which the premises of solely random noise sources is valid is at 4.0 Km. The best fit to a straight line in the 4.0 Km occurs in the low rate domain (i.e. values on the r axis approaching zero).

Predicted Rate vs. Events Observed at 2.5 km



Predicted Rate vs. Events Observed at 3.0 km

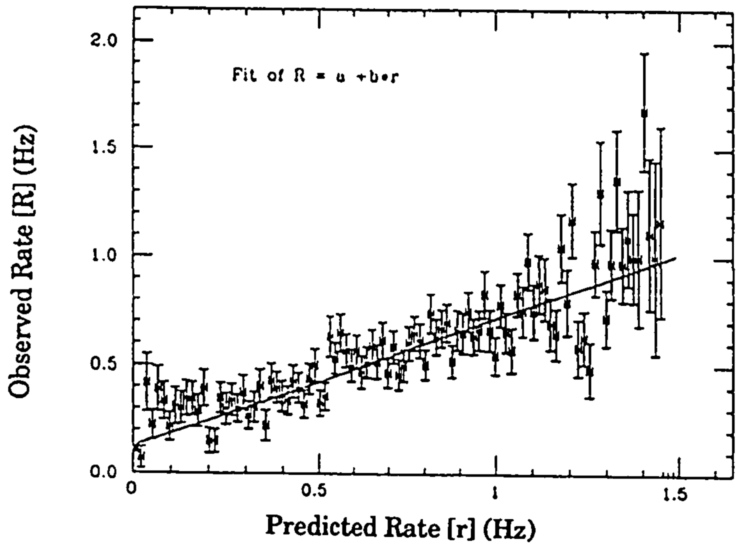
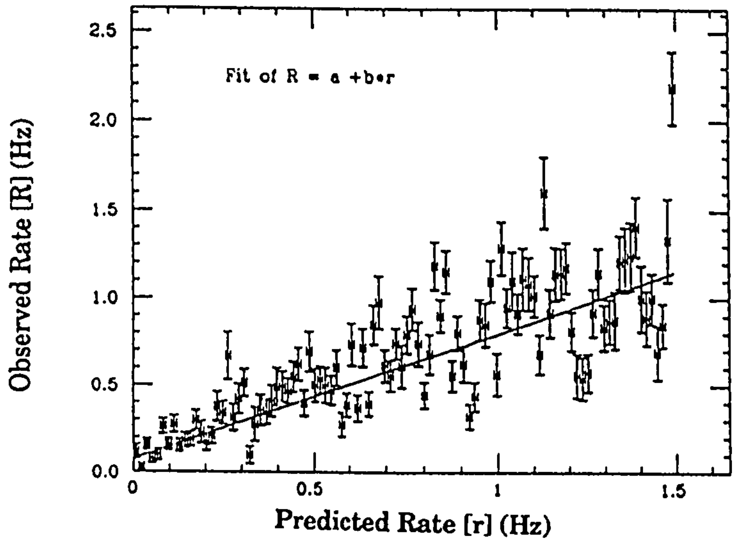


Figure 11.7a The real rates vs. predicted rates plots computed for the first two of the four depths of 2.5 Km and 3.0 Km respectively.

Predicted Rate vs. Events Observed at 3.5 km



Predicted Rate vs. Events Observed at 4.0 km

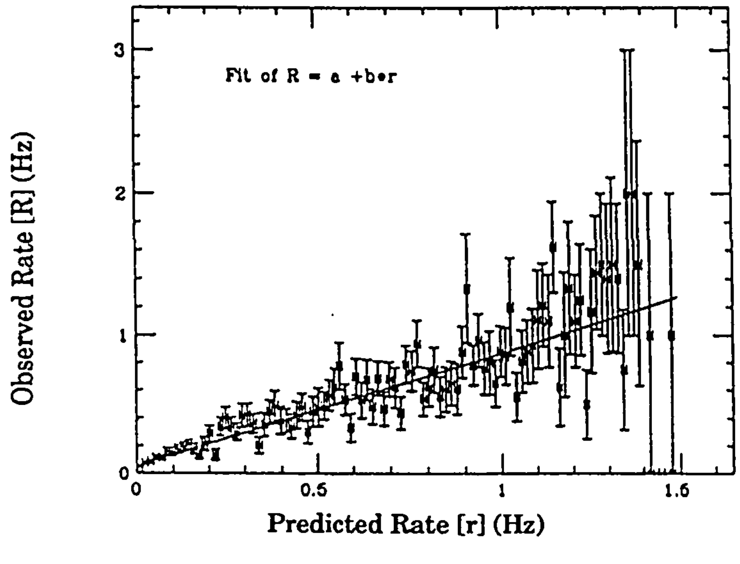


Figure 11.7b The real rates vs. predicted rates plots computed for the last two of the four depths of 3.5 Km and 4.0 Km respectively.

11.2 The Results of the Analysis

Table 11.3 lists all of the data files from the SPS experiment that are not contaminated with calibration pulses. These files are the only possible candidates for this kind of analysis:

Table 11.3
The SPS files that are free of contamination from calibration module pulses.

File Name	Total Events	Trigger Level	No. of Tubes	Depth (Km)
SPS4003	49	unknown	7	2.0
SPS4007	55142	5	7	2.5
SPS4008	934	5	7	2.5
SPS06001	5039	5	6	3.0
SPS08004	1933	5	7	3.5
SPS11008	1153	5	7	4.0

Only four of these files lend themselves to some analysis. SPS4003 with only 49 total events and an unknown trigger level is useless. SPS4007 results in a read error (inside the "Unpack" routine) when trying to extract the raw scalar information but SPS4008 with the same depth and trigger level analyzes just fine. All of these files have 7 active tubes except SPS06001 (the 3.0 km data). To analyze this, a modified program RATEHST6, using the same algorithm detailed above, was written.

Since our premise of randomness of source has been demonstrated in the actual data to be violated everywhere except at 4.0 Km, it is to the data from that depth alone that we turn to for detailed analysis. Figure 11.8 shows the distribution of cosmic ray rates measured by the "excess" method described above by the cut off at different upper bounds in the predicted rate.

The error bars reflect the error in the fit reported out by the least squares fitting program. As the cut off is reduced, the observed cosmic ray rate continuously drops off with essentially the same error until a minimum is reached. Beyond this point the error bars become ridiculous. The result is a rate of

$$R = (2.06 \pm 0.68) \times 10^{-2}\ \mathrm{Hz} \tag{11.13}$$

which compares to the rate determined[1] using a track fitting program of

$$R = 2.09 \times 10^{-2}\ \mathrm{Hz} \tag{11.14}$$

so that internal self-consistency between the two approaches is accomplished.

[1]by Shigenobu Matsuno

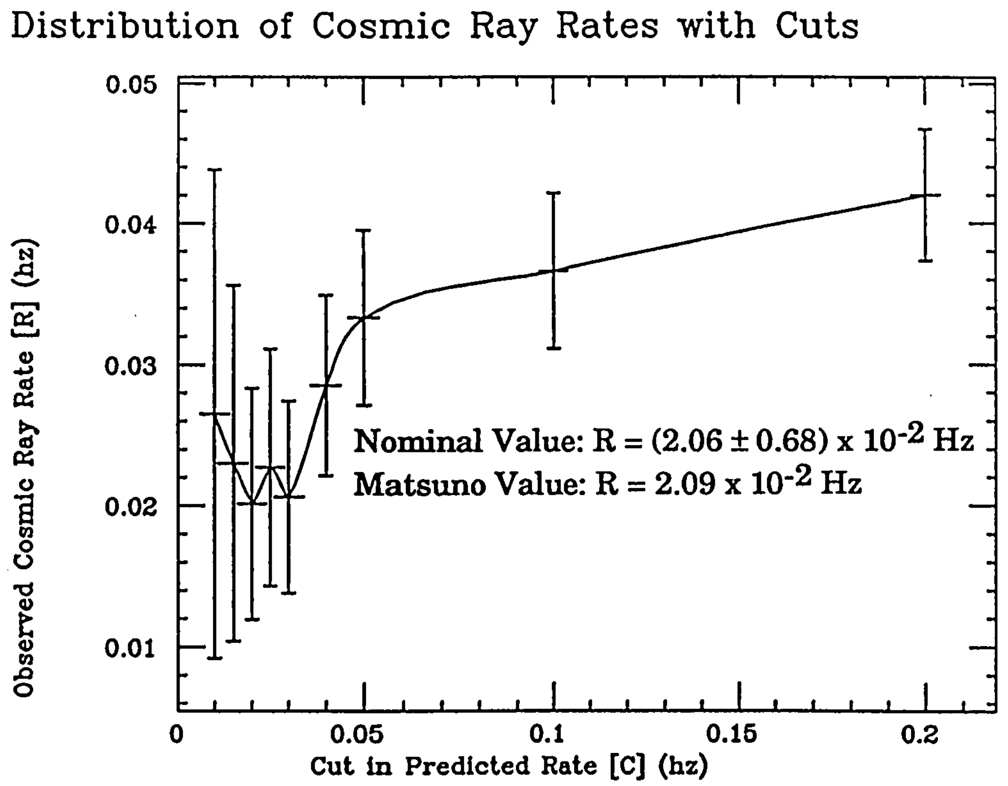


Figure 11.8 The distribution of Cosmic Ray rates at a depth of 4.0 Km plotted as a variation in the cut made in the predicted rate. Note the optimum cut (0.03 Hz) is taken at the point just before the error bars generated by the plot fitting program start to become unreasonable.

The associated plots of integration time, event count, and observed rate vs. the predicted rate all with the optimal upper cut of 0.03 Hz on the predicted rate make up Figures 11.9, 11.10, and 11.11 respectively.

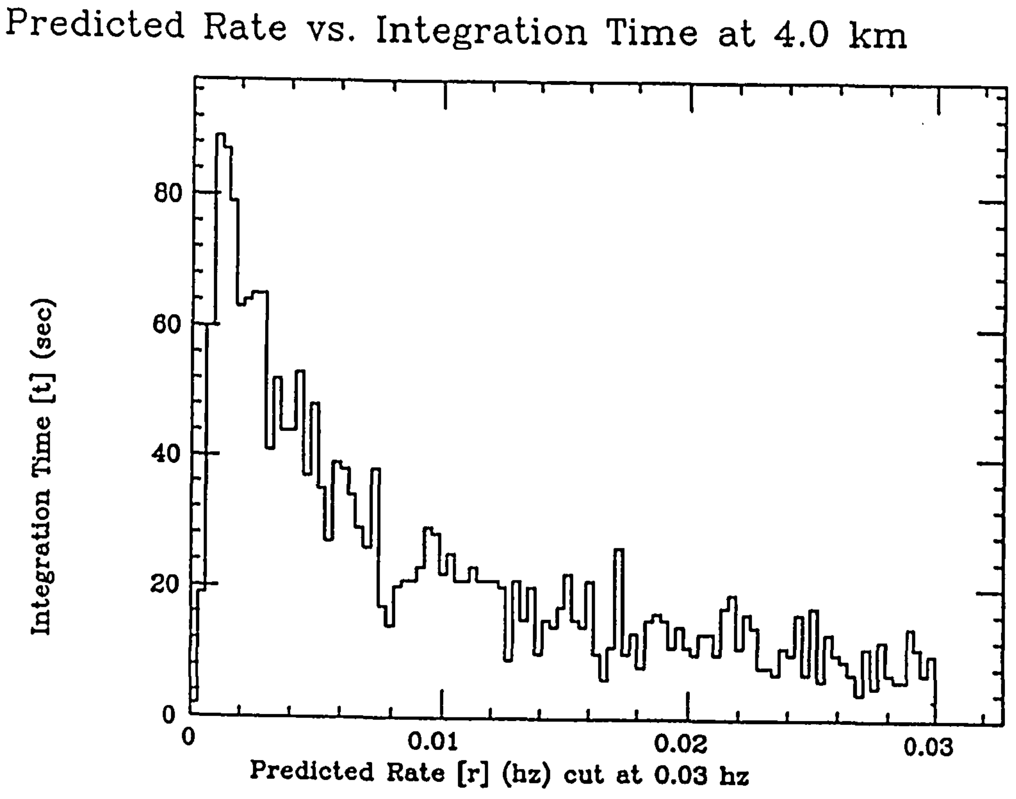


Figure 11.9 The (integration) time vs. predicted rate histogram computed for 4.0 Km with an upper cut of 0.03 Hz in the predicted rate.

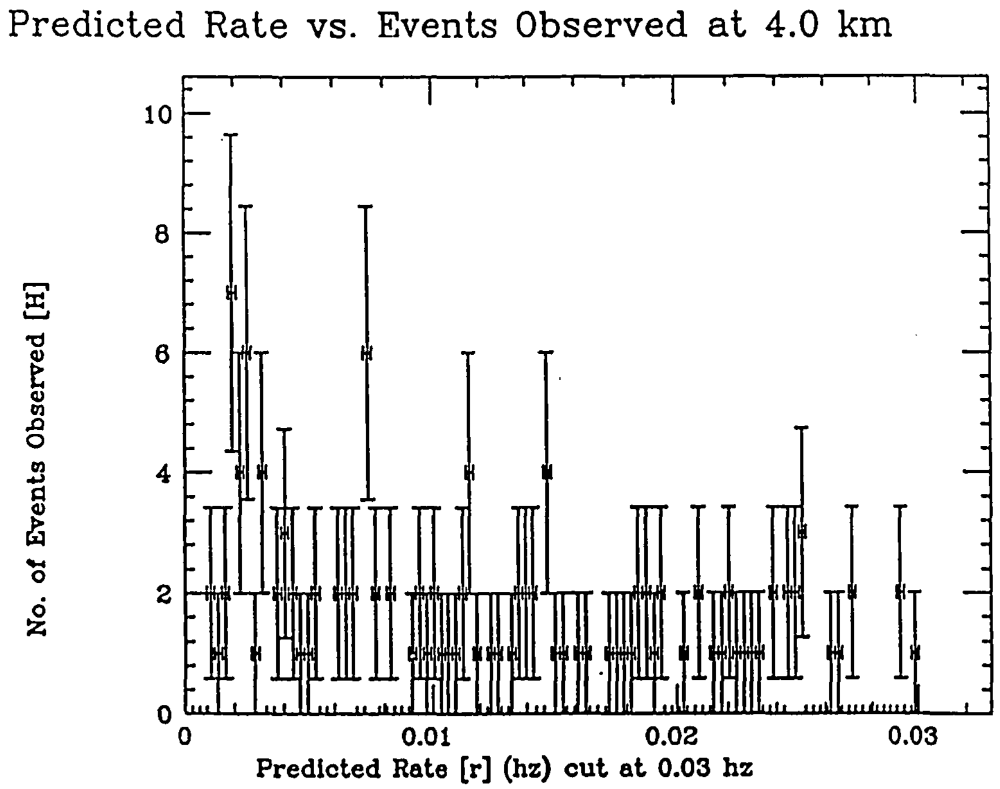


Figure 11.10 The hits (events) vs. predicted rate histogram computed for 4.0 Km with an upper cut of 0.03 Hz in the predicted rate.

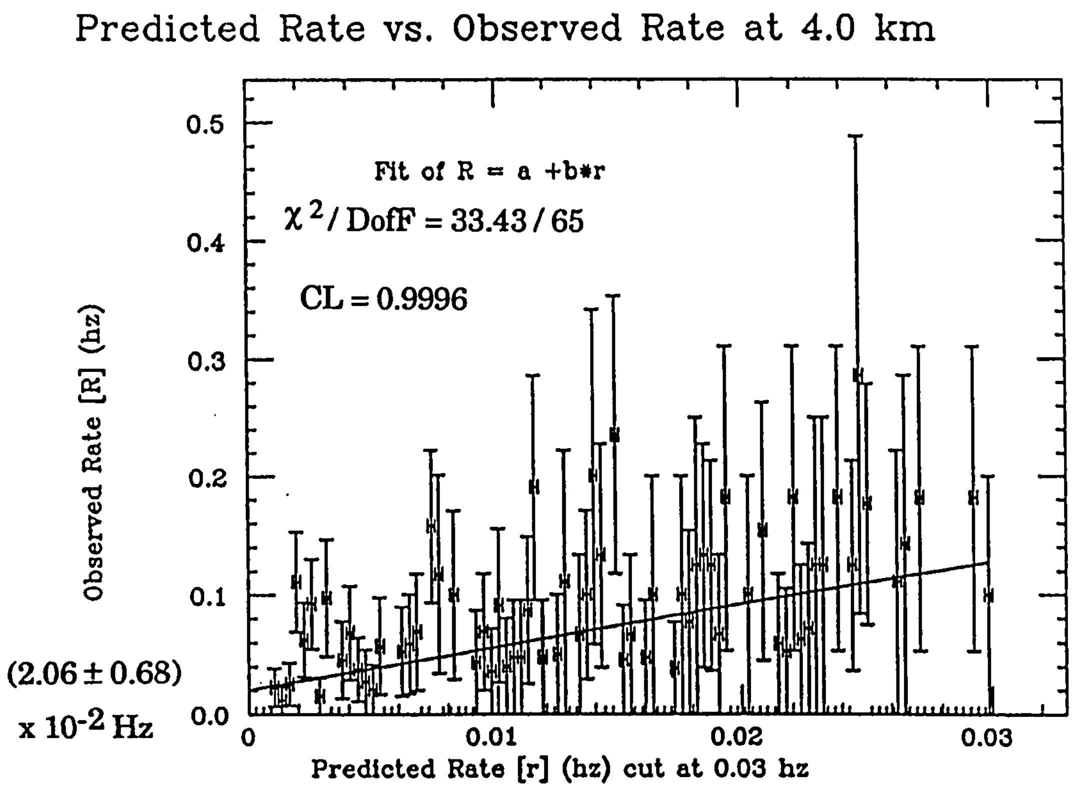


Figure 11.11 The real rate vs. predicted rate histogram computed for 4.0 Km with an upper cut of 0.03 Hz in the predicted rate.

Table 11.4 Comparison of Fitted[2] and Statistical Rates

Depth (Km)	No. of Data Points [n]	Time (sec) [t]	Efficiency of fit [e]	Fitted rate (Hz) [r_s]	Statistical Cosmic Ray rate (Hz) [r_j]
2.0	15	177	0.846	10.0×10^{-2}	
2.5	401	6870	0.846	6.90×10^{-2}	
3.0	53	1108	0.764	6.26×10^{-2}	
3.5	15	395	0.846	4.49×10^{-2}	
4.0	133	7508	0.846	**2.09×10^{-2}**	
4.0	125	7238			**$(2.06 \pm 0.68) \times 10^{-2}$**

Finally, an estimate of the effective area, A_{ef}, of the SPS may be made by assuming an estimate[3] at 4.0 Km depth for the vertical muon flux of 7×10^{-5} m^{-2} s^{-1} sr^{-1}

$$A_{eff} = 3 \pm 1 \times 10^{2} \text{ m}^{2}.$$

11.3 The Determination of the Power Index of the Cosine of the Zenith Angle

The relative intensity with zenith angle can be expressed in the form of

$$I_{\theta} = I_0 \cos^{n} \theta \qquad (11.15)$$

[2] again, the fitted rates are the work of Shigenobu Matsuno

[3] Kobaykawa (1987)

where I_θ represents the intensity at a zenith angle θ and I_0 is the vertical intensity. It is assumed that the atmosphere is flat and as such this representation is limited to zenith angles less than 75^0. The value assigned to the exponent n depends on the data fit. Figure 11.12 shows a compilation of the world's data for a variety of experiments[4].

[4] Crookes, J.N. and Rastin, B.C. (1973)

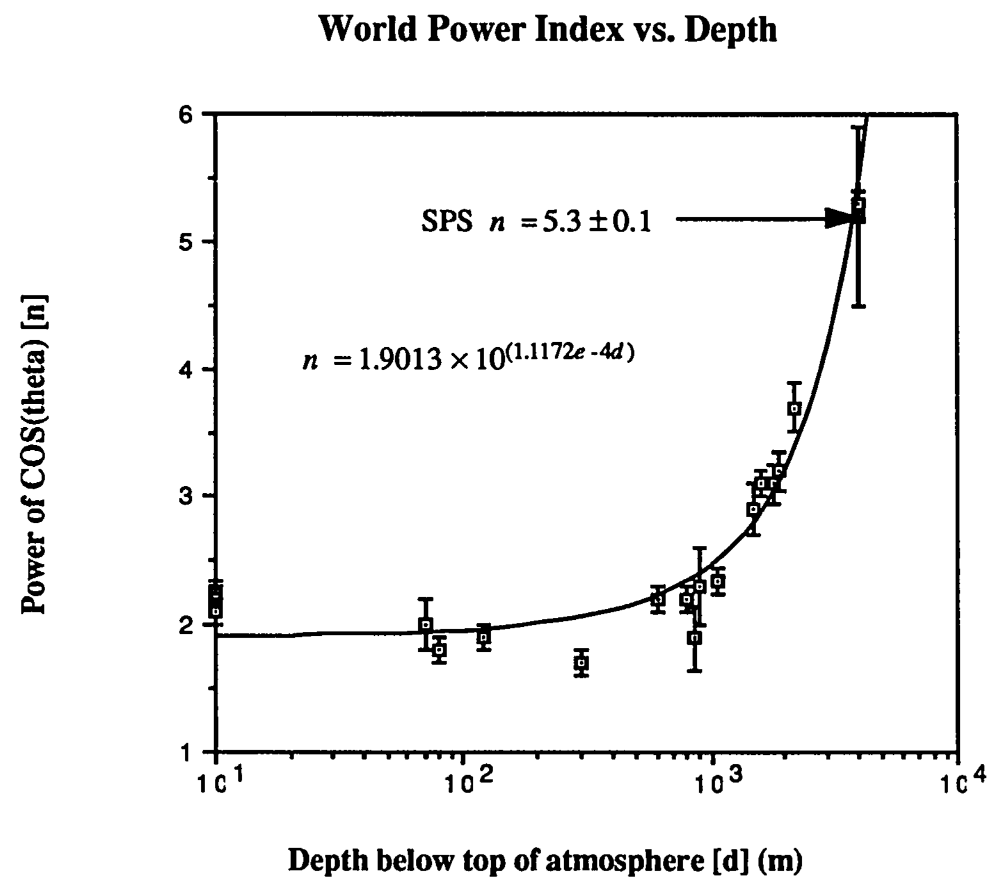


Figure 11.12 The variation of exponent n with depth after the world compilation of Crookes, J.N. and Rastin, B.C. Note that the DUMAND data point of $n = 5.3 \pm 0.1$, which is also plotted here, is consistent with the world experience and greatly reduces the error at 4.0 Km depth. The equation is a best fit exponential.

The exponent n for the SPS experiment was determined by comparing Monte Carlo[5] "fake" events with real fitted events as shown in

[5] written by Victor Stenger

Figure 11.14. The Monte Carlo was used to generate events dependent upon a chosen exponent n. In turn, these events were fed to a 5-fold trigger simulator program[6] to generate "fake" 5–fold events. Then these were filtered by the fitter[7] to select out causally connected "fake" events x_F. Likewise, "real" events were fed to the same fitter to select out causally connected "real" events x_R. Finally, a maximum likelihood procedure was used to compare the "real" and "fake" cases. The SPS is a sparse data set and thus characterized by a Poisson distribution which has a maximum likelihood given by the extreama:

$$ln(p) = \sum_{i=1}^{10} [x_R \, ln(x_F) - x_F] + \text{"}constant\text{"} \tag{11.16}$$

Varying the exponent n resulted in a number of comparisons with the best fit determining the value of n as shown in Figure 11.13. A value of

$$n = 5.3 \pm 0.1 \tag{11.17}$$

results from the exercise.

[6] written by John Clem

[7] written by Shigenobu Matsuno

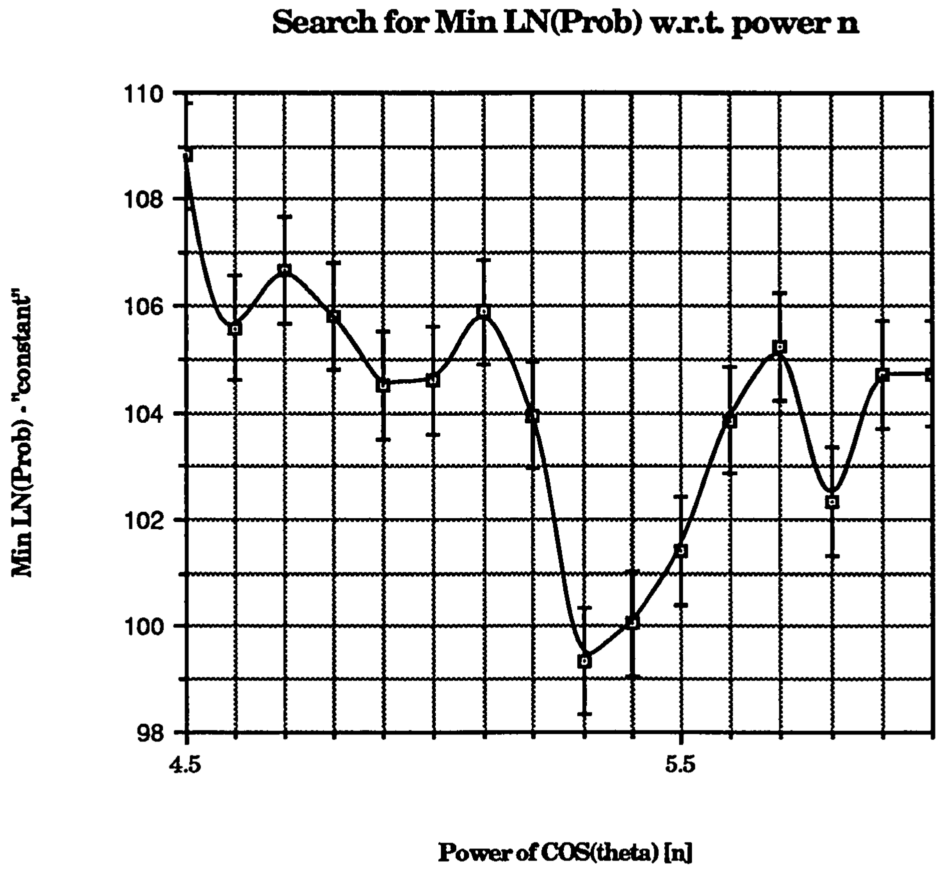


Figure 11.13 The variation of the power exponent n of the cosine of the zenith angle at a depth of 4.0 Km, found by comparing in a maximum likelihood procedure causually fitted "fake" Monte Carlo generated events with "real" events, results in a steep minimum located at $n = 5.3 \pm 0.1$. The curve is a simple interpolation, the error in the y-axis simply reflects a 1% uncertainty.

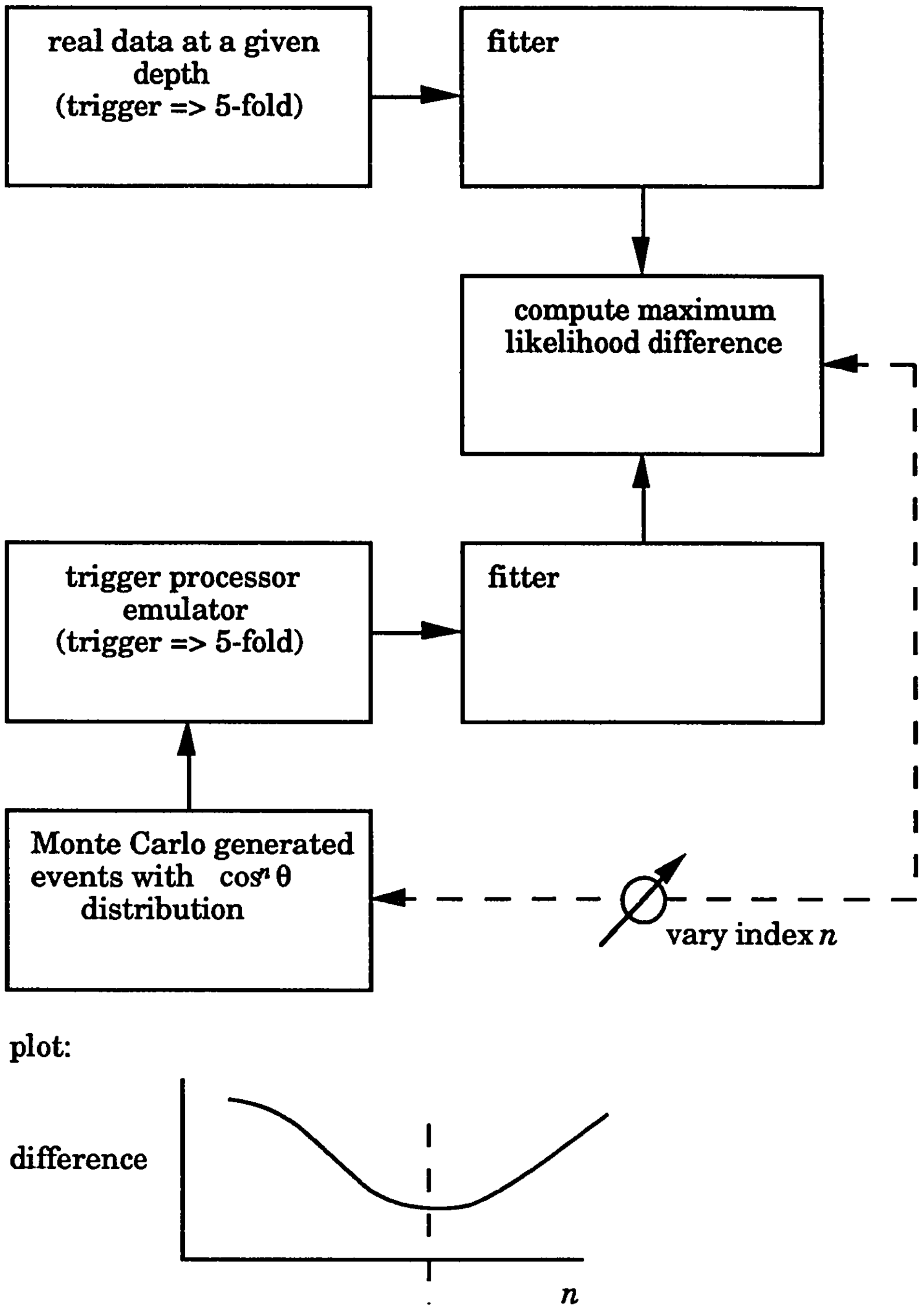

real data at a given depth (trigger => 5-fold)
fitter
compute maximum likelihood difference
trigger processor emulator (trigger => 5-fold)
fitter
Monte Carlo generated events with cos^n θ distribution
vary index n
plot:
difference
n

Figure 11.14 Flow chart outlining the procedure used in determining the power index n of the $\cos^n \theta$ distribution at a depth of 4 Km.

APPENDIX A

Decay-product Correlation of $\tau^+ \tau^-$ Production from $e^+ e^-$ Annihilation

PHYSICAL REVIEW D VOLUME 26, NUMBER 9 1 NOVEMBER 1982

Decay-product correlation of $\tau^+\tau^-$ from e^+e^- annihilation

J. Babson and Ernest Ma

Department of Physics and Astronomy, University of Hawaii at Manoa, Honolulu, Hawaii 96822

(Received 6 July 1982)

We analyze the $\pi^+\pi^-$ angular correlation for the process $e^+e^- \rightarrow \tau^+\tau^-$ with subsequent decays $\tau^+ \rightarrow \pi^+\bar{\nu}_\tau$ and $\tau^- \rightarrow \pi^-\nu_\tau$ as a function of the ν_τ mass as well as the proportion of $V+A$ to $V-A$ coupling strengths.

The discovery of a third charged lepton τ with its accompanying neutrino ν_τ has contributed to a revival of interest in the questions of why nature repeats itself and whether or not neutrinos have mass. The former has to do with the strength and structure of the τ-ν_τ coupling, and the question is whether or not they are identical to those of e-ν_e and μ-ν_μ. Experimentally, pure $V+A$ is certainly ruled out by the observation of the electron momentum spectrum in $\tau \rightarrow \nu_\tau e \nu_e$.[1] However, a small mixture of $V+A$ into the predominant $V-A$ interaction is still possible. As for the coupling strength, a recent measurement[2] of the τ lifetime is indeed consistent with the prediction of generation universality, but a better fit is obtained with a somewhat longer τ lifetime, which is in agreement with a recently proposed[3] gauge model of generation nonuniversality. Assuming a pure $V-A$ interaction, an upper limit of 250 MeV on the ν_τ mass has also been obtained[1] from the electron momentum spectrum.

Of all the decay modes of the τ, the $\pi\nu_\tau$ mode[4,5] is the simplest in kinematics, and by measuring the π momentum, the same upper limit of 250 MeV on the ν_τ mass is now also obtained.[5] In this paper, we consider yet another possible method of determining the ν_τ mass as well as the proportion of $V+A$ to $V-A$ coupling strengths. The idea is to use the $\pi^+\pi^-$ angular correlation for the process $e^+e^- \rightarrow \tau^+\tau^-$ with subsequent decays $\tau^+ \rightarrow \pi^+\bar{\nu}_\tau$ and $\tau^- \rightarrow \pi^-\nu_\tau$.

As the $\tau^+\tau^-$ pair is produced via a virtual photon, the spin-dependent differential cross section is readily calculated.[6] Let the z axis be defined along the direction of the τ^- momentum; then the e^- momentum can be chosen to lie in the x-z plane, making an angle of θ with the z axis. Let E be the energy of the e^- (e^+) beam, then the τ^- (τ^+) is produced with a velocity $\beta=(1-m_\tau^2/E^2)^{1/2}$, and

$$\frac{d\sigma}{d\Omega}(\vec{s},\vec{s}\,') = \frac{\alpha^2}{16E^2}\beta[(2-\beta^2\sin^2\theta) + s_z s_z'(2\cos^2\theta+\beta^2\sin^2\theta)$$
$$+ s_x s_x'(2-\beta^2)\sin^2\theta - s_y s_y'\beta^2\sin^2\theta + (s_z s_x' + s_x s_z')(1-\beta^2)^{1/2}\sin 2\theta] \ , \qquad (1)$$

where $\vec{s}$ ($\vec{s}\,'$) refers to the polarization of the τ^- (τ^+) in its rest frame, and α is the electromagnetic fine-structure constant.

In the decay $\tau^- \to \pi^- \nu_\tau$, let the interaction be a mixture of $V-A$ and $V+A$, i.e., $(1-\gamma_5)+\epsilon(1+\gamma_5)$; then the spin-dependent decay rate is given by

$$\frac{d\Gamma}{d\Omega} = \left(\frac{G_F{}^2 f_\pi{}^2\cos^2\theta_C}{64\pi^2}\right) m_\tau{}^3(1-a^2)(1+a+\eta)^{1/2}(1+a-\eta)^{1/2}(1-a+\eta)^{1/2}(1-a-\eta)^{1/2}(A+B\hat{s}\cdot\hat{p}_\pi) \ , \qquad (2)$$

where $a = m_\pi/m_\tau$, $\eta = m_\nu/m_\tau$, $\hat{p}_\pi$ is a unit vector along the direction of the π^- momentum, and

$$A = (1-a^2)^{-1}\{(1+\epsilon^2)[(1-\eta^2)^2 - a^2(1+\eta^2)] + 4a^2\epsilon\eta\} \ , \qquad (3)$$

$$B = (1-a^2)^{-1}(1-\epsilon^2)(1-\eta^2)(1+a+\eta)^{1/2}(1+a-\eta)^{1/2}(1-a+\eta)^{1/2}(1-a-\eta)^{1/2} \ . \qquad (4)$$

The analogous expression for $\tau^+ \to \pi^+\bar{\nu}_\tau$ is obtained by replacing B with $-B$.

Combining Eqs. (1) and (2), and integrating out the dependence on the production angle θ, we find

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{16\pi^2}\int dx\, dy\, d\phi_1 d\phi_2 \delta(u + v\cos(\phi_1-\phi_2) - z)$$
$$\times\left\{1 - \frac{1}{3-\beta^2}\left(\frac{B}{A}\right)^2 \{xy(1+\beta^2) + (1-x^2)^{1/2}(1-y^2)^{1/2}[(1-\beta^2)\cos(\phi_1-\phi_2) + \cos(\phi_1+\phi_2)]\}\right\} \ , \qquad (5)$$

where $x = \cos\theta_1$, $y = \cos\theta_2$, $z = \cos\theta_{\rm corr}$, with $\theta_{\rm corr}$ the angle between the π^+ and π^- detected in coincidence. Let

$$\beta' = \beta[1 + 4a^2(1+a+\eta)^{-1}(1+a-\eta)^{-1}(1-a+\eta)^{-1}(1-a-\eta)^{-1}]^{1/2} \ ,$$

$$w_1 = [(x+\beta')^2 + (1-\beta^2)(1-x^2)]^{1/2}, \quad w_2 = [(y-\beta')^2 + (1-\beta^2)(1-y^2)]^{1/2} \ ,$$

then the functions u and v in Eq. (5) are given by

$$u = \frac{(x+\beta')(y-\beta')}{w_1 w_2} , \tag{6}$$

$$v = \frac{(1-\beta^2)(1-x^2)^{1/2}(1-y^2)^{1/2}}{w_1 w_2} . \tag{7}$$

Fixing x and y, we then use the δ function to integrate out ϕ_1 and ϕ_2, resulting in

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{4\pi}\int_{-1}^{1} dx \int_{y_1}^{y_2} dy \,[v^2-(u-z)^2]^{-1/2} \times \left[1-\frac{1}{3-\beta^2}\left(\frac{B}{A}\right)^2 F\right] , \tag{8}$$

where

$$F = w_1 w_2 z + \beta'^2 + \beta'(x-y) + \beta^2 xy , \tag{9}$$

and y_1, y_2 (with $y_1 < y_2$ by convention) are the roots of $v^2-(u-z)^2=0$ as functions of z and x.

In the limit of $\beta=0$, we can do the integration in Eq. (8) exactly, and obtain

$$\frac{1}{\sigma}\frac{d\sigma}{dz} = \frac{1}{2}\left[1-\frac{1}{3}\left(\frac{B}{A}\right)^2 z\right] , \tag{10}$$

which shows indeed the expected correlation with $\theta_{\text{corr}} = 180°$ $(z=-1)$ as the most likely angle between the π^+ and π^-. Obviously, the largest effect occurs at the largest value of $(B/A)^2$, which is, in fact, unity and holds for $\epsilon=\eta=0$, i.e., a pure $V-A$ interaction with $m_\nu=0$. For $\epsilon=0$ and $\eta \ll 1$, we have

$$\frac{B}{A} = 1 - \frac{2a^4}{(1-a^2)^2}\eta^2 , \tag{11}$$

which is almost certainly undetectable since $a=0.08$ and η is known to be less than 0.14. Hence this method is very insensitive for determining the value of m_ν. On the other hand, it may be useful for limiting ϵ. For $\eta \ll 1$ and ϵ small, $(B/A)^2 = 1-4\epsilon^2$.

As β increases from zero to one, the correlation becomes even greater at $z=-1$ because of the

Lorentz boosts given to the τ^+ and τ^- along the z axis in opposite directions. The double integral in Eq. (8) must now be computed numerically. On the other hand, if we define

$$z' = xy + (1-x^2)^{1/2}(1-y^2)^{1/2}\cos(\phi_1 - \phi_2) \ , \qquad (12)$$

then we can again do the integration exactly and obtain

$$\frac{1}{\sigma}\frac{d\sigma}{dz'} = \frac{1}{2}\left[1 - \frac{1}{3}\left(\frac{B}{A}\right)^2 z'\right] \ , \qquad (13)$$

which is identical in form to Eq. (10), as expected.

The remaining question is whether z' can be measured in the laboratory. Since $\theta_{1,2}$ and $\phi_{1,2}$ are all defined with respect to the $\tau^+\tau^-$ axis, we must find this axis. This can be done as follows. Let the measured $\pi^+\pi^-$ momenta be $\overline{p}_1 = (p_{1x}, 0, p_{1z})$ and $\overline{p}_2 = (0, 0, p_{2z})$. Let the $\tau^+\tau^-$ axis be the new z axis obtained by two rotations, first an angle ϕ about z, then an angle θ about the new x axis. (The third Euler rotation is not necessary because it will not change the direction of the new z axis.) In terms of this new set of axis,

$$\overline{p}_1 = (p_{1x}\cos\phi, -p_{1x}\cos\theta\sin\phi + p_{1z}\sin\theta, \\ p_{1x}\sin\theta\sin\phi + p_{1z}\cos\theta)$$

and

$$\overline{p}_2 = (0, p_{2z}\sin\theta, p_{2z}\cos\theta) \ .$$

Now we boost $\overline{p}_1$ and $\overline{p}_2$ back to the τ^+ and τ^- rest frames, respectively. Since $\tau \to \pi\nu$ is a two-body decay, the absolute value of the π momentum in the τ rest frame is fixed, hence there will be two equations, one for each π momentum, expressed in terms of θ and ϕ. We solve for θ and ϕ, thus finding the $\tau^+\tau^-$ axis.

This work was supported in part by the U.S. Department of Energy under Contract No. DE-AM03-76SF00235.

[1]W. Bacino *et al.*, Phys. Rev. Lett. 42, 749 (1979).
[2]G. J. Feldman *et al.*, Phys. Rev. Lett. 48, 66 (1982).
[3]X. Li and E. Ma, Phys. Rev. Lett. 47, 1788 (1981).
[4]G. Alexander *et al.*, Phys. Lett. 78B, 162 (1978); W. Bacino *et al.*, Phys. Rev. Lett. 42, 6 (1979).
[5]C. A. Blocker *et al.*, Phys. Lett. 109B, 119 (1982).
[6]Y. S. Tsai, Phys. Rev. D 4, 2821 (1971).

APPENDIX B

Angular Correlations and the Tau-Neutrino Mass

Z. Phys. C - Particles and Fields 20, 5-7 (1983)

Angular Correlations and the Tau-Neutrino Mass

J. Babson and Ernest Ma
Department of Physics and Astronomy, University of Hawaii at Manoa, Honolulu, HI 96822, USA

Received 25 March 1983

Abstract. Effects of a nonzero mass for the tau neutrino ν_τ as well as a right-handed charged-current contribution to the $\tau - \nu_\tau$ coupling are discussed. Angular correlations of the decay products of $\tau^+\tau^-$ in e^+e^- annihilation are calculated as functions of m_{ν_τ} and the relative amount of right-handed $\tau - \nu_\tau$ coupling.

All fundamental fermions, i.e. quarks and leptons, appear to fall into groups which behave exactly like one another except for their different masses. The similarity between the first two generations, i.e. the u and d quarks and the electron and its neutrino, and the c and s quarks and the muon and its neutrino, are now well established. The third generation is presumably comprised of the t and b quarks and the tau and its neutrino. However, the t quark is yet to be discovered, and there are still things about the third charged lepton τ and its neutrino ν_τ, which are not known with certainty. Obviously, more information is needed if one is to be reasonably sure of the validity of generation universality. One directly accessible measurement which is relevant to this question is the τ lifetime. First-round results from the Mark II detector [1], the MAC detector [2], both at the PEP e^+e^- collider at SLAC, and the CELLO detector [3] at the PETRA e^+e^- collider at DESY are 4.6 ± 1.9, 4.9 ± 2.0, and $4.7^{+3.9}_{-2.9}$ respectively in units of 10^{-13}s. A new measurement [4] by the Mark II group, using a new high-precision drift chamber as a vertex detector, has now obtained the value $3.31 \pm 0.57 \pm 0.60 \times 10^{-13}$s, where the first error is the statistical error and the second is the systematic.

To compare with the theoretical expectation, we first assume that all decay products are negligible in mass compared with the decaying particle. Then the τ lifetime is given by

$$t_\tau = t_\mu \left(\frac{m_\mu}{m_\tau}\right)^5 B(\tau \to e\bar{\nu}_e\nu_\tau) = 2.8 \pm 0.2 \times 10^{-13}\text{s}, \qquad (1)$$

where the branching fraction for τ decaying into electron is taken to be $17.6 \pm 1.1\%$, from the most recent and precise measurement [5]. If we use instead the branching fraction for τ decaying into μ, then we must take into account the nonzero mass of the muon, even if we still assume ν_τ to be massless. The decay rate is now suppressed by a factor

$$\frac{\Gamma(\xi)}{\Gamma(0)} = 1 - 8\xi - 12\xi^2 \ln\xi + 8\xi^3 - \xi^4 = 0.973, \qquad (2)$$

where $\xi = m_\mu^2/m_\tau^2$, which corresponds exactly to the measured [5] branching fraction of $17.1 \pm 1.1\%$. Hence $e-\mu$ universality is established to within an accuracy of a few % in τ decay, and the τ lifetime of $2.8 \pm 0.2 \times 10^{-13}$s is an unambiguous prediction of the standard model as long as ν_τ is assumed massless.

The measured values [1–4] of the τ lifetime quoted previously are certainly consistent with the above theoretical prediction, but they are even more consistent with a somewhat higher value. We can take this as a hint that generation universality may only be an approximate symmetry, in which case there are important implications for B-meson physics as well as for W and Z physics [6]. We can also attribute the difference to a nonzero ν_τ mass [7]. For $\tau \to e\bar{\nu}_e\nu_\tau$, since m_e is still negligible, the decay rate is suppressed by the analog of (2), i.e.

$$\frac{\Gamma(\eta)}{\Gamma(0)} = 1 - 8\eta - 12\eta^2 \ln \eta + 8\eta^3 - \eta^4, \tag{3}$$

where $\eta = m_{\nu_\tau}^2/m_\tau^2$. Note that (2) and (3) are identical in form, thanks to the invariance of the $V-A$ interaction under a Fierz transformation. For $\tau \to \mu\bar{\nu}_\mu\nu_\tau$, we must keep both m_μ and m_{ν_τ} nonzero. The suppression factor is now

$$\frac{\Gamma(\xi,\eta)}{\Gamma(0,0)} = f(\xi,\eta)[1 - 7(\xi+\eta) + 12\xi\eta$$
$$-7(\xi^2+\eta^2) - 7\xi\eta(\xi+\eta) + \xi^3 + \eta^3\}$$

$$+ 12\xi^2 \ln\left[\frac{(1-\eta)^2 - 2\xi\eta + \xi^2 + (1+\xi-\eta)f(\xi,\eta)}{2\xi}\right]$$
$$+ 12\eta^2 \ln\left[\frac{(1-\xi)^2 - 2\xi\eta + \eta^2 + (1-\xi+\eta)f(\xi,\eta)}{2\eta}\right]$$
$$- 12\xi^2\eta^2 \ln\left[\frac{1-2(\xi+\eta)+\xi^2+\eta^2+(1-\xi-\eta)f(\xi,\eta)}{2\xi\eta}\right], \tag{4}$$

where

$$f(\xi,\eta) = [1 - 2(\xi+\eta) - 2\xi\eta + \xi^2 + \eta^2]^{1/2} \tag{5}$$

It is clear that (4) is symmetric in ξ and η as required, and a little algebra will show that it reduces to (2) and (3) for $\eta = 0$ and $\xi = 0$ respectively. If m_{ν_τ} is as large as 250 MeV [8, 9], then (3) is numerically equal to 0.861, which gives a 16% enhancement in the theoretical predicted τ lifetime, i.e. 3.27×10^{-13}s, and (4) is numerically equal to 0.836, which in conjunction with (2) again gives the same result. In fact, the ratio of (4) to (3) is more or less numerically given by (2). In other words, the ratio of the μ to e branching fractions is rather insensitive to the ν_τ mass and remains very near 0.97 from $m_{\nu_\tau} = 0$ to 250 MeV. Therefore, measuring the ν_τ mass is not only very important on its own right, but also very necessary for understanding the τ lifetime.

The most direct way of determining m_{ν_τ} is by measuring the electron momentum distribution [8] in $\tau \to e\bar{\nu}_e\nu_\tau$ or the pion momentum [9] in $\tau \to \pi\nu_\tau$. Recently [10], we have also discussed using the $\pi^+\pi^-$ angular correlation in $e^+e^- \to \tau^+\tau^-$ with subsequent decays $\tau^+ \to \pi^+\bar{\nu}_\tau$ and $\tau^- \to \pi^-\nu_\tau$. Unfortunately, the dependence of m_{ν_τ} turns out to be insignificant, so although the angular correlation itself is probably observable, it offers no real hope of seeing the effect of a nonzero m_{ν_τ}. In this report, we complete our analysis with details on other possible angular correlations, which will turn out to depend more sensitively on m_{ν_τ}, but since the total effect is small, the possible indication of a nonzero m_{ν_τ} will still be very difficult to extract.

In $e^+e^- \to \tau^+\tau^-$, let the z-axis be defined along the direction of the τ^- momentum, then the e^- momentum can be chosen to lie in the $x-z$ plane, making an angle of θ with the z-axis. Let E be the energy of the $e^-(e^+)$ beam, then the $\tau^-(\tau^+)$ is produced with a velocity $\beta=(1-m_\tau^2/E^2)^{1/2}$, and the spin-dependent differential cross section via a virtual photon is given by [11]

$$\frac{d\sigma}{d\Omega}(s,s')=\frac{\alpha^2}{16E^2}\beta[(2-\beta^2\sin^2\theta) + s_z s_z'(2\cos^2\theta+\beta^2\sin^2\theta)+s_x s_x'(2-\beta^2)\sin^2\theta - s_y s_y'\beta^2\sin^2\theta+(s_z s_x'+s_x s_z')(1-\beta^2)^{1/2}\sin 2\theta], \tag{6}$$

where s(s') refers to the polarization of the $\tau^-(\tau^+)$ in its rest frame, and α is the electromagnetic fine-structure constant. In the decay $\tau \to x +$ anything, where the particle x is singled out for detection, the spin-dependent decay rate is necessarily of the form [11]

$$\Gamma(\tau\to x)=A+B\hat{s}\cdot\hat{p}_x, \tag{7}$$

where $\hat{p}_x$ is a unit vector along the direction of the x momentum. Consider now only the two-body decays of the τ, i.e. $\pi\nu_\tau$, $K\nu_\tau$, $\rho\nu_\tau$, and $K^*\nu_\tau$. Because of the two-body kinematics, it is possible [10] to use measurements of the momenta (magnitudes and directions) of the observable decay products to reconstruct the original $\tau^-\tau^+$ axis, thereby enabling us to define the correlation angle in terms of quantities expressed in the τ^- and τ^+ rest frames. Let $\tau^-\to x_1$ and $\tau^+\to x_2$, then the angular correlation between x_1 and x_2 is given by [10]

$$\frac{1}{\sigma}\frac{d\sigma}{dz'}=\frac{1}{2}\left[1+\frac{z'}{3}\left(\frac{B}{A}\right)_1\left(\frac{B}{A}\right)_2\right], \tag{8}$$

where

$$z'\equiv\cos\theta_1\cos\theta_2+\sin\theta_1\sin\theta_2\cos(\varphi_1-\varphi_2), \tag{9}$$

and $\theta_{1,2}$ and $\phi_{1,2}$ are the polar and azimuthal angles of the $x_{1,2}$ momenta in the $\tau^{\mp}$ rest frames respectively. By using only two-body decays, we gain the advantage of being able to combine data at different energies in comparing with (8).

It is also clear from (8) that if a certain angular correlation is to be observable, the corresponding (B/A)'s have to be reasonably large. Let us assume that the $\tau - \nu_\tau$ coupling is a mixture of $V - A$ and $V + A$, i.e. $(1 - \gamma_5) + \varepsilon(1 + \gamma_5)$, then for $\tau^- \to \pi^- \nu_\tau$ or $K^- \nu_\tau$,

$$\frac{B}{A} = \frac{(1 - \varepsilon^2)(1 - \eta)\left[1 - \frac{2(1 + a)\eta - \eta^2}{(1 - a)^2}\right]^{1/2}}{(1 + \varepsilon^2)\left[1 - \frac{(2 + a)\eta - \eta^2}{(1 - a)}\right] + \frac{4a\varepsilon\eta^{1/2}}{1 - a}}, \qquad (10)$$

where $\eta = m_{\nu_\tau}^2/m_\tau^2$ as before, and $a = m_\pi^2/m_\tau^2$ or m_K^2/m_τ^2 as appropriate. For $\tau^+ \to \pi^+ \bar{\nu}_\tau$ or $K^+ \bar{\nu}_\tau$, B/A simply reverses its sign. In the limit $\varepsilon = 0$ and $\eta = 0$, $B/A = 1$ in (10); hence the $\pi^- \pi^+$ angular correlation peaks in the backward direction as expected, with $\sigma^{-1} d\sigma/dz'$ equal to $\frac{2}{3}$ at $z' = -1$ and $\frac{1}{3}$ at $z' = +1$. This backward-forward ratio of 2 to 1 should not be too difficult to see experimentally. Unfortunately, the dependence on m_{ν_τ} is very weak [10], so that for small η,

$$\frac{B}{A} = \frac{1 - \varepsilon^2}{1 + \varepsilon^2}$$

$$\cdot \left[1 - \frac{4a\varepsilon\eta^{1/2}}{(1 - a)(1 + \varepsilon^2)} - \frac{2a^2\eta}{(1 - a)^2} + \frac{16a^2\varepsilon^2\eta}{(1 - a)^2(1 + \varepsilon^2)^2}\right], \qquad (11)$$

which is almost indistinguishable from $(1 - \varepsilon^2)/(1 + \varepsilon^2)$ because a is numerically small.

For $\tau^- \to \rho^- \nu_\tau$ or $K^* \nu_\tau$, the situation is quite different. Since a vector boson is now in the final state, the usual helicity argument that a massless ν_τ must be produced opposite in direction to the τ^- polarization is no longer valid. Hence the angular correlation is expected to be less. A straight forward calculation shows that in this case,

$$\frac{B}{A} = \left(\frac{1-2a}{1+2a}\right) \cdot \frac{(1-\varepsilon^2)\left(1 - \frac{\eta}{1-2a}\right)\left[1 - \frac{2(1+a)\eta - \eta^2}{(1-a)^2}\right]^{1/2}}{(1+\varepsilon^2)\left[1 - \frac{(2-a)\eta - \eta^2}{(1-a)(1+2a)}\right] - \frac{12a\varepsilon\eta^{1/2}}{(1-a)(1+2a)}}. \tag{12}$$

In the limit $\varepsilon = 0$ and $\eta = 0$, $B/A = (1-2a)/(1+2a)$, which for $a = m_\rho^2/m_\tau^2$ is equal to 0.46; hence $\sigma^{-1} d\sigma/dz'$ for $\rho^-\rho^+$ is given by $\frac{1}{2}(1 - 0.07z')$, which has a backward-forward ratio of only 1.15. On the other hand, the dependence on m_{ν_τ} is much more significant than in the $\pi^-\pi^+$ case.

For small η,

$$\frac{B}{A} = \left(\frac{1-2a}{1+2a}\right)\left(\frac{1-\varepsilon^2}{1+\varepsilon^2}\right)\left[1 + \frac{12a\varepsilon\eta^{1/2}}{(1-a)(1+2a)(1+\varepsilon^2)} - \frac{2a(4-7a)\eta}{(1-a)^2(1-4a^2)} + \frac{144a^2\varepsilon^2\eta}{(1-a)^2(1+2a)^2(1+\varepsilon^2)^2}\right], \tag{13}$$

which amounts to a 7% effect in the $\rho^-\rho^+$ angular correlation for $m_{\nu_\tau} = 250$ MeV. The same analysis is also applicable to τ decaying into the A_1.

We now come to the three-body leptonic decays $\tau^- \to e^- \nu_\tau \bar{\nu}_e$ and $\tau^- \to \mu^- \nu_\tau \bar{\nu}_\mu$. Since the $\tau^-\tau^+$ axis cannot be determined precisely from kinematics, the angular correlation must be defined at each specific energy. The e^- or μ^- momentum is also not fixed in magnitude as in the two-body case, hence an extra integration is required as well. For $\tau^- \to \mu^- \nu_\tau \bar{\nu}_\mu$ and normalizing to $\varepsilon = \eta = \zeta = 0$, the spin-independent part of the decay rate is given by

$$
\begin{aligned}
A = {} & f(\xi,\eta)\{(1+\varepsilon^2)[1-7(\xi+\eta)+12\xi\eta \\
& -7(\xi^2+\eta^2)-7\xi\eta(\xi+\eta)+\xi^3+\eta^3] \\
& -4\varepsilon\eta^{1/2}[1-5\xi+10\eta-2\xi^2-5\xi\eta+\eta^2]\} \\
& +12\xi^2(1+\varepsilon^2-2\varepsilon\eta^{1/2}) \\
& \cdot\ln\left[\frac{(1-\eta)^2-2\xi\eta+\xi^2+(1+\xi-\eta)f(\xi,\eta)}{2\xi}\right] \\
& +12\eta[(1+\varepsilon^2)\eta+2\varepsilon\eta^{1/2}(1-2\xi+\eta)] \\
& \cdot\ln\left[\frac{(1-\xi)^2-2\xi\eta+\eta^2+(1-\xi+\eta)f(\xi,\eta)}{2\eta}\right] \\
& -12\xi^2\eta[(1+\varepsilon^2)\eta-2\varepsilon\eta^{1/2}] \\
& \cdot\ln\left[\frac{1-2(\xi+\eta)+\xi^2+\eta^2+(1-\xi-\eta)f(\xi,\eta)}{2\xi\eta}\right], \qquad (14)
\end{aligned}
$$

where $f(\xi,\eta)$ is defined in (5). The spin-dependent part is even more complicated, i.e.

$$
\begin{aligned}
B = & -\tfrac{1}{3}[(1-\xi^{1/2})^2-\eta] \\
& \cdot\Big\{(1-\xi^{1/2})^3[1+5\xi^{1/2}+15\xi+3\xi^{3/2}] \\
& -\eta(1-\xi^{1/2})[11+31\xi^{1/2}+57\xi+21\xi^{3/2}] \\
& -\frac{\eta^2}{(1-\xi^{1/2})}[47-5\xi^{1/2}-15\xi+21\xi^{3/2}]-3\eta^3\Big\} \\
& -4\eta^2[3(1-\xi^2)+2\eta]\ln\left[\frac{(1-\xi^{1/2})^2}{\eta}\right] \\
& +4\varepsilon\eta^{1/2}\Big\{[(1-\xi^{1/2})^2-\eta](1-\xi^{1/2})^2(1+4\xi^{1/2}+\xi) \\
& +10\eta(1+\xi^{1/2}+\xi)+\eta^2] \\
& -6\eta[(1-\xi)^2+\eta(1+\xi)]\ln\left[\frac{(1-\xi^{1/2})^2}{\eta}\right]\Big\} \\
& -\varepsilon^2\Big\{[(1-\xi^{1/2})^2-\eta][(1-\xi^{1/2})^4(1+6\xi^{1/2}+\xi) \\
& -\eta(1-\xi^{1/2})^2(7+26\xi^{1/2}+7\xi) \\
& -\eta^2(7+2\xi^{1/2}+7\xi)+\eta^3]+12\eta^2(1-\xi)^2 \\
& \cdot\ln\left[\frac{(1-\xi^{1/2})^2}{\eta}\right]\Big\}. \qquad (15)
\end{aligned}
$$

In the above, we have integrated out the magnitude of the μ^- momentum in the τ^- rest frame. Hence (14) and (15) only apply at the production threshold energy, i.e. $E = m_\tau$. In the limit $\varepsilon = \eta = 0$, $B/A = -\frac{1}{3}$ for e^- and -0.341 for μ^-, hence the $\mu^- e^+$ angular correlation, for example, is rather small, with a backward-forward ratio of only 1.08. For $m_{\nu_\tau} = 250$ MeV, $(B/A)_{e^+}(B/A)_{\mu^-}$ decreases by about 12%.

In conclusion, effects of a nonzero ν_τ mass can be significant in many processes: 16% in the total τ lifetime, 7% in the $\rho^- \rho^+$ angular correlation, and 12% in the $\mu^- e^+$. More precise experimental information on m_{ν_τ} is clearly desirable.

Acknowledgements. One of us (E.M.) thanks M. Delfino, S. Pakvasa, and S.F. Tuan for discussions. This work was supported in part by the U.S. Department of Energy Under Contract No. DE-AM03-76SF00235.

References

1. G.J. Feldman et al.: Phys. Rev. Lett. 48, 66 (1982)
2. W.T. Ford et al.: Phys. Rev. Lett. 49, 106 (1982)
3. H.J. Behrend et al.: DESY Report No. 82-056 (August 1982)
4. J.A. Jaros: SLAC Report No. 2992 (October 1982)
5. C.A. Blocker et al.: Phys. Rev. Lett. 49, 1369 (1982)
6. X. Li, E. Ma: Phys. Rev. Lett. 47, 1788 (1981)
7. L. Maiani: in Proc. of the XXIst International Conference on High Energy Physics (1982), to be published
8. W. Bacino et al.: Phys. Rev. Lett. 42, 749 (1979)
9. C.A. Blocker et al.: Phys. Lett. 109B, 119 (1982)
10. J. Babson, E. Ma: Phys. Rev. D26, 2497 (1982)
11. Y.S. Tsai: Phys. Rev. D4, 2821 (1971)

APPENDIX C

The Feynman Rules for Q.E.D.

C.1 Notation[1]

Diagrams are labeled as follows:

p_i - external momenta (lines)
s_i - external spins
q_i - internal momenta (lines)

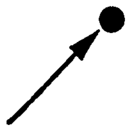
Arrows are placed on the external fermion lines (see Figure C.1 below) indicating whether the fermion is a particle or an antiparticle. Arrows on internal fermion lines meerly indicate that the direction of flow in the diagram is retained (i.e. every vertex must have one arrow entering and one arrow exiting). Arrows on external photon lines point "forward" while arrows on internal photon lines are arbitrary.

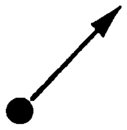
[1] for example, see Griffiths

C.2 Labeling of External Lines

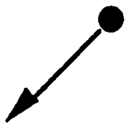
Each external line contributes factors as follows:

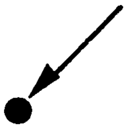
particles:

incoming  u

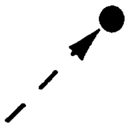
outgoing  $\bar{u}$

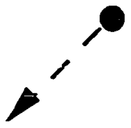
antiparticles:

incoming  $\bar{v}$

outgoing  v

photons:

incoming  ε^{μ}

outgoing  $\varepsilon^{\mu *}$

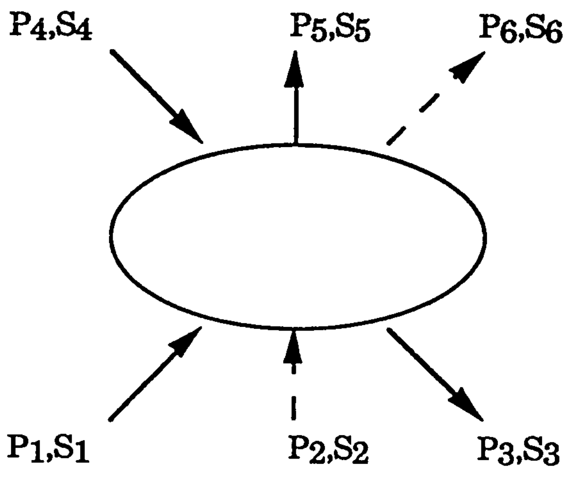


Figure C.1 A typical Q.E.D. diagram with the external lines shown and labeled.

C.3 Representation of Vertices

Each vertex contributes a factor of $ig_e\gamma^\mu$ where the dimensionless coupling constant g_e is related to the charge of the lepton by

$$g_e = e\sqrt{4\pi / \hbar c} = \sqrt{4\pi\alpha} \tag{C.1}$$

Often, the Heaviside-Lorentz units with

$$\hbar = c = 1 \tag{C.2}$$

are used in which case the charge of the positron is written as

$$g_e = e \tag{C.3}$$

Instead, the Guassian unit system is used so that all factors of $\hbar$ and c are retained with the dimensionless coupling constant being

$$\alpha = \frac{e^2}{\hbar c} \approx \frac{1}{137} \tag{C.4}$$

Generalizing, the Q.E.D. coupling constant is

$$g_{QED} = -q\sqrt{4\pi / \hbar c} \tag{C.5a}$$

where q is the charge of the particle (not the antiparticle) so that for

leptons (e, m, t) $\quad q = -e \quad$ (C.5b)

and for

"up" quarks (u, c, t) $\quad q = +\frac{2}{3}e \quad$ (C.5c)

C.4 Propagators and Calculation Rules

Each internal line supplies a factor of

leptons

$$P(q) = \frac{i\,(\gamma^\mu q_\mu + m_l c)}{q^2 - m^2{}_l c^2} \tag{C.6a}$$

photons

$$P^{\mu\nu}(q) = \frac{-i\, g^{\mu\nu}}{q^2} \tag{C.6b}$$

where m_l is the mass of the lepton.

The calculation rules are as follows:

RULE 1 - Conservation of Energy and Momentum

Each vertex supplies a factor in the form of a delta function

$$(2\pi)^4\, \delta^4(k_1 + k_2 + k_3) \tag{C.7}$$

where the k's are the four-momenta coming into the vertex. If the arrow is outgoing, then k is minus the four-momenta of that line (with the convention completely reversed for antiparticles). This factor forces the conservation of momentum-energy at the corresponding vertex.

RULE 2 - Integrate over Internal Momenta

Each internal line contributes a factor of

$$\frac{1}{(2\pi)^4} d^4 q_i \tag{C.8}$$

and integrate over all internal momenta.

RULE 3 - Cancel the Residual Delta Function

The final result of the integration will include a delta function of the form

$$(2\pi)^4 \delta^4(p_1 + p_2 + \ldots - p_n) \tag{C.9}$$

forcing an overall conservation of momentum-energy. Erase this factor and what is left is iM where M is the amplitude of the diagram. In practice, one writes down all the diagrams contributing to the process up to the desired order of calculation and finds the amplitude for each one. A total amplitude of all contributing diagrams is then found which is then used to calculate the appropriate cross section or lifetime by way of the Golden Rules.

RULE 4 - Antisymmetrization

With fermions, one has both particles and antiparticles. If two diagrams differ only in the interchange of two incoming (or outgoing) leptons (anti-leptons) or one incoming lepton and one outgoing anti-lepton, then the corresponding amplitudes are combined with a minus sign. Otherwise, they are combined with a plus sign.

C.5 Golden Rule for Decays

Should a particle 1 decay into a series of other particles 2, 3, 4, ..., n

$$1 \dashrightarrow 2+3+4+\dots\ +n \tag{C.10}$$

then the dcay rate is given by

$$d\Gamma = |M|^2 \frac{S}{2\hbar m_1}\left[\left(\frac{c\,d^3\mathbf{p}_2}{(2\pi)^3\,2E_2}\right)\left(\frac{c\,d^3\mathbf{p}_3}{(2\pi)^3\,2E_3}\right)\cdots\left(\frac{c\,d^3\mathbf{p}_n}{(2\pi)^3\,2E_n}\right)\right]$$
$$\times(2\pi)^4\,\delta^4(p_1-p_2-p_3\,\dots-p_n) \tag{C.11}$$

where $p_i=(E_i/c,\mathbf{p}_i)$ is the four-momenum of the ith particle carrying mass m_i so that $E_i^2-\mathbf{p}_i^2c^2=m_i^2c^4$. The delta functions enforce the conservation of energy and momentum and is zero unless

$$p_1=p_2+p_3+p_4+\dots+p_n \tag{C.12}$$

and the decaying particle is assumed to be at rest so that

$$p_1=(m_1c,\mathrm{o}) \tag{C.13}$$

Finally, S is the product of statistical factors $1/j!$ for each group of j identical particles in the final state. Typically, one is not interested in the outgoing momenta of the decay products and thus one integrates over all outgoing momenta to obtain the total decay rate Γ. For the special case of only two final decay products, the Golden Rule for Decay reduces to

$$\Gamma = \frac{S}{\hbar m_1}\left(\frac{c}{4\pi}\right)^2\frac{1}{2}\int\frac{|M|^2}{E_2E_3}\,\delta^4(p_1-p_2-p_3)\,d^3\mathbf{p}_2\,d^3\mathbf{p}_3 \tag{C.14}$$

Here one has the advantage that the amplitude

$$M = M(\mathbf{p}_2, \mathbf{p}_3) \tag{C.15}$$

can be explicitely integrated to yield the expression

$$\Gamma = \frac{S\,|\mathbf{p}|}{8\pi\hbar m_1^2 c}|M|^2 \tag{C.16a}$$

where $|\mathbf{p}|$ is the magnitude of either outgoing momentum which in terms of the masses is

$$|\mathbf{p}| = \frac{c}{2m_1}\sqrt{m_1^4 + m_2^4 + m_2^4 - 2m_1^2m_2^2 - 2m_1^2m_3^2 - 2m_2^2m_3^2} \tag{C.16b}$$

C.6 Golden Rule of Scatterings

For scattering, say of particles 1 anbd 2 yielding particles 3, 4, ..., n

$$1 + 2 \dashrightarrow 3 + 4 + \ldots + n \qquad \text{(C.17)}$$

then the cross section is given by

$$d\sigma = |M|^2 \frac{\hbar^2 S}{4\sqrt{(p_1 . p_2)^2 - (m_1 m_2 c^2)^2}} \left[\left(\frac{c d^3 \mathbf{p}_3}{(2\pi)^3 2E_3} \right) \left(\frac{c d^3 \mathbf{p}_4}{(2\pi)^3 2E_4} \right) \cdots \left(\frac{c d^3 \mathbf{p}_n}{(2\pi)^3 2E_n} \right) \right]$$
$$\times (2\pi)^4 \delta^4 (p_1 + p_2 - p_3 - p_4 \ldots - p_n)$$

(C.18)

where $p_i = (E_i/c, \mathbf{p}_i)$ is the four-momenum of the ith particle carrying mass m_i so that $E_i^2 - \mathbf{p}_i^2 c^2 = m_i^2 c^4$ as before. The delta functions enforce the conservation of energy and momentum and is zero unless

$$p_1 + p_2 = p_3 + p_4 + \ldots + p_n \qquad \text{(C.19)}$$

and S is the product of statistical factors $1/j!$ for each group of j identical particles in the final state. Equation (C.18) yields the cross section for an interaction whete the three-momentum of particle 3 lies in the range $d\mathbf{p}_3$ about the value $\mathbf{p}_3$, 4 in the range $d\mathbf{p}_4$ about $\mathbf{p}_4$, etc. Typically, one seeks to study only the angle 3 emerges integrating over all the remaining final state momenta ($\mathbf{p}_4$, $\mathbf{p}_5$, ..., $\mathbf{p}_n$) and the magnitude of $\mathbf{p}_3$. This yields $d\sigma / d\Omega$ as the "differential" cross section for scattering 3 into solid angle Ω.

For the special case of only two decay products, the Golden Rule for Scattering reduces to

$$d\sigma = \left(\frac{\hbar c}{8\pi}\right)^2 \frac{S}{\sqrt{(p_1 \cdot p_1)^2 - \left(m_1 m_2 c^2\right)^2}} \frac{|M|^2}{E_3 E_4} \delta^4 \left(p_1 + p_2 - p_3 - p_4]\right) d^3\mathbf{p}_3 d^3\mathbf{p}_4$$

(C.20)

which in the CM frame where

$$\sqrt{(p_1 \cdot p_2)^2 - \left(m_1 m_2 c^2\right)^2} = (E_1 + E_2)|\mathbf{p}_1|/c \tag{C.21}$$

reduces to

$$\frac{d\sigma}{d\Omega} = \left(\frac{\hbar c}{8\pi}\right)^2 \frac{S|M|^2}{(E_1 + E_2)^2} \frac{|\mathbf{p}_f|}{|\mathbf{p}_i|} \tag{C.22}$$

where $|\mathbf{p}_i|$ is the magnitude of either incoming momentum and $|\mathbf{p}_f|$ is the magnitude of either outgoing momentum.

APPENDIX D

Source Code Listing of the Executive Program for Controlling the String Optical and Calibration Modules (SOM)

The source code for program SOM consists of 23 modules which are listed below roughly in a top down fashion with the highest level routines being listed first and the lowest level or direct hardware driving modules listed last. This pattern approximates the order in which modules are called in the program and is listed according to the order shown in Table D.1 below:

Table D.1 Listing Order of SOM Routines

Class	Name	Description
Main:		
	SOM	Main program
	INITIAL	Initialize stack and 300 baud serial port
Communications control:		
	CREPLY	Echo back valid command
	RREPLY	Send back single's rate value
	SYNCH	Synchronize on first byte of command string
Parser:		
	CMDPOLL	Parse the command string for a valid command

Table handler:

ANAPOLL	Poll all (16) analog channels
ANLKUP	Look up (read) analog value from table for specified channel
UPDATE	Update (write) analog value to table for specified channel

Command executive:

CMDSERV	Service the command request
RRATE	Read the PMT single's rate value

Device drivers:

ATOD	Drives AtoD converter for specified channel
DTOA	Drives DtoA converter for specified channel
MODID	Look up this module's identification number
SETDSCR	Set the threshold of the discriminator
SETHV	Set the PMT's high voltage

Error handling:

CMDERR	Report a command error
DEVERR	Report a device error
FORERR	Report a format error

Glue:

ISALNO	Check if byte is alphanumeric
ISHEX	Check if byte is a hexidecimal number
RAMPDN	Ramp down the PMT's high voltage
RAMPUP	Ramp up the PMT's high voltage

The listing of the individual modules now follows:

```
$DEBUG

NAME STRING_OPTICAL_MODULE

; program SOM
;
; program is first attempt to control the DUMAND optical module
; with 15 analog input telemetry channels and two output channels,
; namely the PMT high voltage level and the discriminator threshold
; level as well as a singles rate counter.  this is a very elementry
; program taking all control directly from a 300 baud
; terminal sending only printable ASCII characters.

; author - John F. Babson, University of Hawaii Physics
; revision date - Apr. 30, 1986

;
; link references
;
    EXTRN CODE (INITIAL, DTOA, MDELAY, ANAPOLL, SYNCH,
                CMDPOLL, CMDSERV)
   EXTRN CODE (XEQY, XGTY, XNEY)

$INCLUDE(UHPS.INC)
$INCLUDE(TELMTY.INC)

; global definitions and variables
;
; const


DADELAY   EQU   0FFH      ; ~1/4 second delay (255 * 1 ms)
DAMLCLK   EQU   R5        ; ms clock
DAPWRTIM EQU   015H      ; (15H = 20D) 20 x ~1/4 sec = 5 sec
                          ; time out
; var

GLOBAL_DATA     SEGMENT DATA
RSEG   GLOBAL_DATA

        DEV:            DS   1         ; device register (hex)
        CMD:            DS   2         ; command buffer (hex)
        DATUM:          DS   2         ; data buffer (hex)
        ANALOG:         DS   MAXCHAN ; analog look up table (binary)
        OUTCF:          DS   1         ; output communications flag
                                     ;      TRUE ==> output enabled
                                     ;      FALSE ==> output suppressed
        ERRFLG:         DS   1         ; communications error flag
        SYNCFLG:        DS   1         ; communications
                                     ;    synchronization flag

        PUBLIC DEV, CMD, DATUM, ANALOG, OUTCF, ERRFLG, SYNCFLG

; local definitions and variables

; const

; var

SOM_DATA      SEGMENT DATA
RSEG          SOM_DATA

        LOOP:           DS   1         ; infinite loop condition (binary)
        CHAN:           DS   1         ; channel for D to A initialization
        VOLT:           DS   1         ; running voltage value for D to A
        QSCOUNT:        DS   1         ; 1/4 sec counter
```

```
; begin program

; SOM_CODE    SEGMENT CODE
; RSEG   SOM_CODE

CSEG

ORG     0       ; system start up location

SOM:

        CALL        INITIAL    ; initialize  system

        ; hardware  restart  assures  that  microcontroller  takes  control
        ; of the BUSS before D to A chips are powered up thus assuring
        ; that  no  unwanted  voltages  are  written  out.   now,  enter  a
        ; 5  second  loop  writing  out  initial  D  to  A  voltages  to  assure
        ; D  to  A  state  before  proceeding.

        ; initialize  1/4  sec  counter  -  save  it  on  the  stack

        MOV  QSCOUNT, #DAPWRTIM

        ; loop  for  5  sec's  at  1/4  sec  intervals

        %MWHILE(QSCOUNT,MXGTY,#00H,ILOOP1,ILOOP2)

                ; initialize  output  of  PMT  high  voltage  channel  to  ground

                MOV   CHAN, #HIVOLT    ; directly access PMT high voltage
                                       ; channel

                PUSH CHAN

                MOV   VOLT, #00H       ; select  zero  voltage  (ground)  as
                                       ;  initial  state

                PUSH  VOLT             ;  (latched)  state

                CALL  DTOA
        ; initialize output of discriminator threshold level to
        ; highest value, thus cutting off all video output

        MOV  CHAN, #DISCR     ; directly access threshold channel

        PUSH CHAN

        MOV  VOLT, #0FFH      ; select for no video output as
                              ; initial state

        PUSH    VOLT         ; (latched) state

        CALL  DTOA

        ; now, delay for about 1/4 sec before using
        ; D to A channels again

        MOV  DAMLCLK, #DADELAY  ; 0FFH ~1/4 sec

        CALL MDELAY

      DEC    QSCOUNT    ; decrement 1/4 sec counter value
                        ;   to continue the count

; end while

%MWEND(ILOOP1,ILOOP2)

; loop := true

MOV    LOOP,#TRUE

; while (loop = true)

%MWHILE(LOOP,MXEQY,#TRUE,PMT1,PMT2)

        ; loop := true

        MOV    LOOP,#TRUE
```

```
            ; read command line and parse it for command
            %MREPEAT (ERROR1)
                ; loop for first synch byte
                CALL SYNCH
                %MREPEAT (ERROR3)
                ; loop for the remaining bytes
                    CALL CMDPOLL
                %MUNTIL (SYNCFLG,MXNEY,#TRUE,ERROR3,ERROR4)
            %MUNTIL (ERRFLG,MXNEY,#TRUE,ERROR1,ERROR2)
            ; poll all analog devices
            CALL ANAPOLL
            ; service the command
            CALL   CMDSERV
        ; end while
        %MWEND(PMT1,PMT2)

MOS:

END
```

```
NAME INITIAL

; subroutine INITIAL - used to initialize the system hardware
; including the serial I/O port, the system stack, and any
; AtoD and DtoA hardware.
; UNIVERSAL 300 baud version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 20, 1985

;
; link references
;

        PUBLIC  INITIAL

; global definitions and variables
;
; const

$INCLUDE (CONSNT.INC)


; local definitions and variables
;
; const

        ; control of serial port

        BAU300     EQU 99H          ; 300 baud timer (self determined)
        SERPORT    EQU 11011010B    ; Serial Port
```

```
; var

DSEG at     8

STACK:      DS    13    ; at power-up, the stack pointer is
                        ;  initialized
                        ; to point here.  24 byte space reserved.
                        ;     -----> 16 byte is an experiment to permit
                        ; linking without using XDATA memory

; begin subroutine

INITIAL_CODE    SEGMENT CODE
RSEG INITIAL_CODE

        INITIAL:

                ; This is the initializing section. Execution always
                ; starts at address 0 on power-up.

                MOV  TMOD,#MOD21     ; set timer1 mode to auto-reload
                MOV  TH1,#BAU300     ; set timer for 300 BAUD
                MOV  SCON,#SERPORT   ; prepare the Serial Port
                SETB TR1             ; start clock

                RET

        LAITINI:

END
```

```
;$DEBUG

NAME COMMAND_REPLY

; subroutine CREPLY - returns a command string with value
; requested of the given channel - STRING version, looks for multiple
; device calls and returns nothing

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 23, 1985

;
; link references
;
    EXTRN CODE (HEXBIN, ANALKUP, BINHEX, PUT_CHAR)
    EXTRN CODE (XEQY)

    PUBLIC CREPLY

; global variables


    EXTRN DATA (CMD, OUTCF)

$INCLUDE(UHPS.INC)


; local definitions and variables
;
; const

        ; note: null used as empty value for upper byte
        ; in hex to binary conversion
; var

CREPLY_DATA   SEGMENT DATA
RSEG    CREPLY_DATA

      HVALUE:     DS     2
      OFFSET:     DS     1
      AVALUE:     DS     1
      C_OUT:      DS     1
      DUMMY:      DS      1        ; dummy ASCII hex cell

; begin  subroutine

CREPLY_CODE   SEGMENT CODE
RSEG    CREPLY_CODE

      CREPLY:

                 ; if (multiple devices called) {

           %MIF (OUTCF,MXEQY,#FALSE,CREPLY1)

                   RET     ; i.e. send back no messages

            ; }

           CREPLY1:

                ; fetch second character of command word

           MOV     DUMMY,#NUM0

           PUSH    DUMMY

           PUSH    CMD+1

           CALL    HEXBIN

           POP     OFFSET
```

```
    ; use it to look up (read) corresponding value in analog
    ;table

PUSH  OFFSET

CALL  ANALKUP

POP   AVALUE

      ; convert binary value to hex for reply transmission

PUSH  AVALUE

CALL  BINHEX

POP   HVALUE

POP   HVALUE+1

     ; begin reply in the following order

MOV   A,#DOLLAR        ; synchronization byte

CALL  PUT_CHAR

MOV   A,CMD            ; first command byte

CALL  PUT_CHAR

MOV   A,CMD+1          ;second command byte

CALL  PUT_CHAR

MOV   A,HVALUE         ; first data byte

CALL  PUT_CHAR

MOV   A,HVALUE+1       ; second data byte

CALL  PUT_CHAR
```

```
            MOV     A,#CR
            CALL    PUT_CHAR        ;<CR>
            RET
        YLPERC:
    END
```

```
$DEBUG

NAME RATE_REPLY

; subroutine rreply - calls rrate which measures the PMT
; pulse repetition rate for 100 ms and then return
; the result in a formated message (reply).

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 23, 1985

; link references

      PUBLIC RREPLY

      EXTRN CODE (PUT_CHAR, BINHEX, RRATE)
      EXTRN CODE (XEQY)

; global constants and variables

$INCLUDE (UHPS.INC)

      EXTRN DATA (OUTCF, CMD)

; local definitions and variables
;
; const
; var

RREPLY_DATA SEGMENT DATA
RSEG RREPLY_DATA

      HVALUE: DS     2      ; hex value buffer

; begin  subroutine

RREPLY_CODE SEGMENT CODE
RSEG RREPLY_CODE

     RREPLY:

               ; if (multiple devices called) {

          %MIF (OUTCF,MXEQY,#FALSE,RREPLY1)

                 RET     ; i.e. send back no messages

          RREPLY1:

               ; measure PMT pulse repetition rate for 100 ms
             ; leaving result in COUNTER0 bytes TL0 and TH0
               ; with overflow condition in bit TF0.

          CALL   RRATE

               ; begin reply in the following order

          MOV    A,#DOLLAR        ; synchronization  byte
          CALL   PUT_CHAR

          MOV    A,CMD            ; first  command  byte
          CALL   PUT_CHAR

          MOV    A,CMD+1          ; second command byte
          CALL   PUT_CHAR
```

```
  ; check for overflow condition and if TRUE
  ; then return error message ("OVER") else
  ; return value

  ; overflow condition of COUNTER0 is indicated
  ; by bit TF0 being set (i.e. high).

MOV    C,TF0

JNC    NOOVER          ; i.e. jump if no overflow

       ; output error condition

    MOV    A,#BIGO      ; "O"
    CALL   PUT_CHAR

    MOV    A,#BIGV      ; "V"
    CALL   PUT_CHAR

    MOV    A,#BIGE      ; "E"
    CALL   PUT_CHAR

    MOV    A,#BIGR      ; "R"
    CALL   PUT_CHAR

    JMP    OVER

NOOVER:

       ; output the rate count in TH0 and TL0
         ; first converting from binary to hex

       ; convert and transmit TH0

    PUSH   TH0
    CALL   BINHEX
    POP    HVALUE
    POP    HVALUE+1

    MOV    A,HVALUE
    CALL   PUT_CHAR
```

```
            MOV     A,HVALUE+1
            CALL    PUT_CHAR

              ; convert and transmit TL0

            PUSH    TL0
            CALL    BINHEX
            POP     HVALUE
            POP     HVALUE+1

            MOV     A,HVALUE
            CALL    PUT_CHAR

            MOV     A,HVALUE+1
            CALL    PUT_CHAR

        OVER:

             ; finish up the messge

            MOV     A,#CR
            CALL    PUT_CHAR        ; <CR>

            RET

    YLPERR:

END
```

```
;$DEBUG

NAME SYNCHRONIZE

; subroutine SYNCH - parses for initial synchronization byte <$>
; in input data stream - any additional synchronization bytes
; are handled in CMDPOL
;   STRING / POWER module version

; author - John F. Babson, University of Hawaii Physics
; revision date - Apr. 30, 1986

;
; link references
;

     PUBLIC SYNCH

     EXTRN CODE (GET_CHAR, XEQY)

$INCLUDE(UHPS.INC)


; local definitions and variables
;
; const
; var

SYNCH_DATA    SEGMENT DATA
RSEG     SYNCH_DATA

     SYNBYTE:     DS       1  ; synchronization byte

; begin  subroutine

SYNCH_CODE    SEGMENT CODE
RSEG     SYNCH_CODE


    SYNCH:

            ; loop until synch byte <$> found

         %MREPEAT (SYNCH1)

               CALL    GET_CHAR

               MOV     SYNBYTE,A

         %MUNTIL (SYNBYTE,MXEQY, #DOLLAR, SYNCH1, SYNCH2)

          RET

    HCNYS:

END
```

```
$DEBUG

NAME COMMAND_POLL

; routine used to poll for a valid command
;   STRING module version

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 26, 1986

; note: unbuffered communications!  each character fetched by "CALL
; GET_CHAR"
; rather than read out of a string buffer.

;
; link references
;

      PUBLIC CMDPOLL

      EXTRN CODE (GET_CHAR, PUT_CHAR, PUT_STRING, PUT_CRLF,
                  XEQY, XNEY)
      EXTRN CODE (ASCBIN, MODID, ISHEX, FORERR, XLTY, XGTY)

; global variables

      EXTRN DATA (DEV, CMD, DATUM, OUTCF, ERRFLG, SYNCFLG)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const
; var

CMDPOL_DATA   SEGMENT DATA
RSEG    CMDPOL_DATA

     CHAR:       DS     1      ; character buffer
     IDENT:      DS     1      ; module identity
     DEVID:      DS     1      ; binary form of DEV
     HFLAG:      DS     1      ; hex condition flag (TRUE means
                               ; value is valid hex character)
     LOGFLG:     DS     1      ; control flag for multiconditional
                               ; logic

; macro definitions

%*DEFINE (MSYNCH(LAB)) LOCAL LABEL

(

     CALL   GET_CHAR

     MOV    CHAR,A

     %MIF (CHAR,MXEQY,#DOLLAR,SYN%LAB)

        MOV    SYNCFLG,#TRUE

        RET

     SYN%LAB:

)
```

```
; begin subroutine

CMDPOL_CODE SEGMENT CODE
RSEG  CMDPOL_CODE

CMDPOLL:

        ; initialize control flags

      ; enable communications output

    MOV    OUTCF,#TRUE

      ; disable error flag

    MOV    ERRFLG,#FALSE

      ; disable message synch flag

    MOV    SYNCFLG,#FALSE


        ; first check for consistent device destination

     %MSYNCH(1)      ; first device byte in DEV

    MOV    DEV,CHAR

    %MSYNCH(2)      ; second device byte in CHAR

    %MIF (DEV,MXNEY,CHAR,CMDPOL1)

        MOV    ERRFLG,#TRUE   ; inconsistent device
                              ; so look for next message
        RET
```

```
CMDPOL1:


  ; second check if message is for this module

 CALL  MODID       ; read module identity hardware

 POP   IDENT       ; returning its identity


 PUSH  DEV         ; convert DEV info to binary

 CALL  ASCBIN      ; form in order to compare

 POP   DEVID       ; it to the device identity


  ; if (devid != ident) {

; MIF (DEVID,MXNEY,IDENT,CMDPOL1A)

           ; either the device is not unique or it is not this
          ; device so shut off all message replies

     MOV    R0,DEVID  ; hand expansion of MIF
     MOV    R1,IDENT  ;   to overcome out of range
     CALL   XNEY      ;   problem

         JC     INTERM      ; to the "if" code
         JMP    CMDPOL1A    ; to the "else" code

      INTERM:  ; the "if" code

     MOV    OUTCF,#FALSE
  ; switch (devid) {

    ; case A or B /* power or instrumentation modules */

 %MIF (DEVID,MXEQY,#0AH,CPCASE1) ; power module

     MOV    ERRFLG,#TRUE

       RET     ; return early

 CPCASE1:

  %MIF (DEVID,MXEQY,#0BH,CPCASE2) ; instrumentation
                                  ; module

     MOV    ERRFLG,#TRUE

       RET     ; return early

 CPCASE2:

    ; case E  /* calibration module "all call" */

%MIF (DEVID,MXEQY,#0EH,CPCASE3)

        ; if (ident == 8 or 9) { break

     %MIF (IDENT,MXEQY,#08H,CPCASE4A)

           JMP    CPCASE4C   ; break

      CPCASE4A:

     %MIF (IDENT,MXEQY,#09H,CPCASE4B)

           JMP    CPCASE4C   ; break
```

```
          ; } else {
        %MELSE (CPCASE4B,CPCASE4C)
              MOV    ERRFLG,#TRUE
                RET    ; return early
          ; }
         CPCASE4C:
               JMP    CMDPOL9 ; break to execute command
     CPCASE3:
         ; case F /* optical module "all call" */
    %MIF (DEVID,MXEQY,#0FH,CPCASE5)
           ; if (ident == 1..7) { break
            MOV    LOGFLG,#TRUE  ; initialize logic flag
          %MIF (IDENT,MXLTY,#01H,CPCASE6A) ; lower bound
              MOV    LOGFLG,#FALSE
               JMP    CPCASE6B  ; break on false
         CPCASE6A:
         %MIF (IDENT,MXGTY,#07H,CPCASE6B)  ; upper bound
              MOV    LOGFLG,#FALSE
               JMP    CPCASE6B  ; break on false
```

```
 CPCASE6B:

        ; test for composite logic condition

%MIF (LOGFLG,MXEQY,#TRUE,CPCASE6C)

            JMP    CPCASE6F  ; okay so break out

 CPCASE6C:

        ; if(ident == C or D) {break /* SBC or spare channel */

%MIF (IDENT,MXEQY,#0CH,CPCASE6D)

            JMP    CPCASE6F  ; break /* SBC */

 CPCASE6D:

%MIF (IDENT,MXEQY,#0DH,CPCASE6E)

            JMP    CPCASE6F  ; break /* spare */

   ; } else {

%MELSE (CPCASE6E,CPCASE6F)

         MOV    ERRFLG,#TRUE

            RET    ; return early

   ; }

 CPCASE6F:

            JMP    CMDPOL9 ; break to execute command
```

```
CPCASE5:

    ; case 0 /* combined optical and calibration
   ;          modules "all call" */

%MIF (DEVID,MXEQY,#00H,CPCASE7A)

         ; if (ident == 1..9) { break

         MOV      LOGFLG,#TRUE  ; initialize logic flag

       %MIF (IDENT,MXLTY,#01H,CPCASE8A)  ; lower bound

            MOV    LOGFLG,#FALSE

              JMP     CPCASE8B  ; break on false

       CPCASE8A:

      %MIF (IDENT,MXGTY,#09H,CPCASE8B)  ; upper bound

            MOV    LOGFLG,#FALSE

              JMP     CPCASE8B  ; break on false

       CPCASE8B:

           ; test for composite logic condition

    %MIF (LOGFLG,MXEQY,#TRUE,CPCASE8C)

              JMP     CPCASE7  ; okay so break out

       CPCASE8C:
```

```
                ; if (ident == C or D) { break /* SBC spare channel */
          %MIF (IDENT,MXEQY,#0CH,CPCASE8D)
                    JMP     CPCASE7  ; break /* SBC */
           CPCASE8D:
          %MIF (IDENT,MXEQY,#0DH,CPCASE8E)
                    JMP     CPCASE7  ; break /* spare */
            ; } else {
          %MELSE (CPCASE8E,CPCASE8F)
                  MOV    ERRFLG,#TRUE
                    RET     ; return early
            ; }
           CPCASE8F:
                   JMP    CMDPOL9 ; break to execute command
       CPCASE7:
        ; } default
      %MELSE (CPCASE7A,CPCASE7B)
          MOV    ERRFLG,#TRUE
            RET     ; return early
       CPCASE7B:
```

```
 ; } else {

%MELSE (CMDPOL1A,CMDPOL9)

          ; putc (0XFF)  /* output hex FF */

          ; output a hex FF in order to start up and
            ; synchronize the string modem receiver circuit

       MOV     A,#0FFH

       CALL    PUT_CHAR

 ; }

CMDPOL9:


  ; third parse message for a command

  %MSYNCH(3)   ; first command byte

MOV     CMD,CHAR

 %MSYNCH(4)   ; second command byte

MOV     CMD+1,CHAR

  %MSYNCH(5)    ; termination or first data byte

MOV     DATUM,CHAR
```

```
 ; if (datum == <CR>) {

%MIF (DATUM,MXEQY,#CR,CMDPOL10)

          ; check if command is an incomplete
           ; write command, if it is return an error

    %MIF (CMD,MXEQY,#BIGW,CMDPOL10A)

            CALL    FORERR

           MOV     ERRFLG,#TRUE

             RET      ; early return

    %MELSE (CMDPOL10A,CMDPOL10B)

            RET      ; message  completed

     CMDPOL10B:

CMDPOL10:

  ; verify valid hex character

PUSH  DATUM

 CALL  ISHEX

POP    HFLAG

%MIF (HFLAG,MXNEY,#TRUE,CMDPOL11)

     CALL   FORERR

     MOV    ERRFLG,#TRUE

      RET
```

```
CMDPOL11:
 %MSYNCH(6)   ; second data byte
 MOV    DATUM+1,CHAR
    ; verify valid hex character
 PUSH  DATUM+1
 CALL  ISHEX
 POP    HFLAG
%MIF (HFLAG,MXNEY,#TRUE,CMDPOL12)
        CALL    FORERR
        MOV     ERRFLG,#TRUE
        RET
CMDPOL12:
   ; check for command termination
 %MSYNCH(7)   ; command terminator in CHAR
 ; if (char == <CR>) {
%MIF (CHAR,MXEQY,#CR,CMDPOL13)
        RET     ; message completed
    CMDPOL13:

       ; message not properly terminated

    CALL   FORERR

    MOV    ERRFLG,#TRUE

    RET

LLOPDMC:

END
```

```
;$DEBUG

NAME ANALOG_POLL

; subroutine ANAPOL - used to poll all analog input channels and
; place their values in the analog telemetry table ANALOG
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

; link references
;
      PUBLIC ANAPOLL

     EXTRN CODE (UPDATE, XGTY)

$INCLUDE(UHPS.INC)
$INCLUDE(TELMTY.INC)

; local definitions and variables
;
; const
; var

ANAPOL_DATA   SEGMENT DATA
RSEG    ANAPOL_DATA

      CHANNEL:        DS      1       ; a to d channel

; begin  subroutine

ANAPOL_CODE   SEGMENT CODE
RSEG    ANAPOL_CODE

ANAPOLL:

        ; point  to  first  channel

     MOV     CHANNEL,#FIRST_CHAN

       ; repeat

     %MREPEAT(APOLL1)

             ; read atod channel and update table

           PUSH    CHANNEL

           CALL    UPDATE

             ; point to next channel

           INC     CHANNEL

        ; until (channel > last_chan)

     %MUNTIL(CHANNEL,MXGTY,#LAST_CHAN,APOLL1,APOLL2)

     RET

LLOPANA:

END
```

```
;$DEBUG

NAME ANALOG_TABLE_LOOKUP

; subroutine ANALKUP - reads values from telemetry table ANALOG
;   SOM verison

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

; link  references
;

; global  variables

        PUBLIC ANALKUP

         EXTRN DATA(ANALOG)           ; telemetry table

; local  definitions  and  variables
;
; const

; var

        AN_TABLE_DATA    SEGMENT DATA
        RSEG     AN_TABLE_DATA

                CHAN:          DS       1         ; pointer offset from table
                                                 ; beginning - a to d channel
                AENTRY:        DS       1         ; entry found in table
                POSITION:      DS       1         ; analog table pointer
                CALK1:         DS       1         ; save address upper byte
                CALK2:         DS       1         ;  "     "    lower   "
```

```
; begin  subroutine

AN_TABLE_CODE    SEGMENT CODE
RSEG     AN_TABLE_CODE

      ANALKUP:

             POP     CALK1   ; save the
              POP     CALK2   ;   return address

               POP     CHAN    ; input channel identity

                  ; point to appropriate position in analog table and fetch
                 ; entry i.e. location := analog + taboff
                 ; entry ---> @location

            MOV     A,#ANALOG        ; point to table entry

            ADD     A,CHAN           ; add offset

            MOV     POSITION,A       ; point to position in table

            MOV     R0,POSITION      ; load pointer

            MOV     A,@R0            ; fetch value in table

            MOV     AENTRY,A         ; write value

               ; return the value

            PUSH    AENTRY

            PUSH    CALK2
            PUSH    CALK1

            RET

      PUKLANA:
END
```

```
;$DEBUG

NAME UP_DATE

; subroutine UPDATE - puts values into telemetry table ANALOG
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

; link references
;
        PUBLIC UPDATE

        EXTRN CODE (ATOD)

; global variables

          EXTRN DATA (ANALOG)      ; telemetry table


; local definitions and variables
;
; const
; var

      AN_TABLE_DATA   SEGMENT DATA
      RSEG    AN_TABLE_DATA

             CHAN:        DS     1      ; a to d channel
             AENTRY:      DS     1      ; value returned on channel
             POSITION:    DS     1      ; analog table pointer
             CUPD1:       DS     1      ; save address upper byte
             CUPD2:       DS     1      ;  "    "   lower  "

; begin  subroutine

AN_TABLE_CODE  SEGMENT CODE
RSEG    AN_TABLE_CODE

UPDATE:

        ; start and read atod

      POP     CUPD1    ; save the

      POP     CUPD2    ;   return address

      POP     CHAN     ; input channel identity

      PUSH    CHAN     ; pass it on to a to d

      CALL    ATOD

      POP     AENTRY

         ; point to appropriate position in analog table and enter
        ; aentry.  i.e. position := analog + chan
        ; @position --> aentry

      MOV     A,#ANALOG             ; point to table entry

      ADD     A,CHAN                ; add offset

      MOV     POSITION,A            ; point to position in table
```

```
        MOV   R0,POSITION           ; load pointer
        MOV   A,AENTRY              ; fetch value
        MOV   @R0,A                 ; write value in table
        PUSH  CUPD2                 ; load the
        PUSH  CUPD1                 ;    return address
        RET
ETADPU:
END
```

```
$DEBUG

NAME COMMAND_SERVICE

; routine services the command or returns an error message for
; invalid commands.  it is here that the various output channels are
; defined.
;    string optical module version

; author - John F. Babson, University of Hawaii Physics
; revision date - May 21, 1986

;
; link references
;
      EXTRN CODE (XEQY, ISHEX, RREPLY, CREPLY, SETHV, SETDSCR,
                  CMDERR)

      PUBLIC CMDSERV

$INCLUDE(UHPS.INC)

; global constants and variables

      EXTRN DATA (CMD)


; local definitions and variables
;
; const

; var

CMD_SERV_DATA     SEGMENT DATA
RSEG    CMD_SERV_DATA

        HEXFLG:    DS    1  ; condition flag for hex value
```

```
; begin subroutine

CMD_SERV_CODE     SEGMENT CODE
RSEG    CMD_SERV_CODE

CMDSERV:

          ; first, check for a read command

         ; if (cmd(0) = 'R') then

      %MIF(CMD,MXEQY,#BIGR,CSERV1)

                  ; command for repetition rate?

            %MIF (CMD+1,MXEQY,#BIGR,CSERV2)

                      CALL    RREPLY          ; reply with rate value

                    RET

             CSERV2:

                 ; command to read a telemetry channel?

             PUSH    CMD+1

               CALL    ISHEX    ; channel must be hex

             POP    HEXFLG

            %MIF (HEXFLG,MXEQY,#TRUE,CSERV3)

                   CALL    CREPLY

                   RET
      CSERV3:

         ; otherwise, command is invalid

      CALL    CMDERR  ; echo back bad command

      RET

  CSERV1:

    ; second, check for a write command

    ; if (cmd(0) = 'W') then

%MIF(CMD,MXEQY,#BIGW,CSERV4)

          ; if (cmd+1 = 'E') then

      %MIF(CMD+1,MXEQY,#BIGE,CSERV5)

              CALL    SETHV

              RET

      CSERV5:

      %MIF(CMD+1,MXEQY,#BIGF,CSERV6)

             CALL    SETDSCR

             RET

      CSERV6:

      CALL    CMDERR ; echo back bad command

      RET
    CSERV4:

      ; default, invalid command

    CALL    CMDERR ; echo back bad command

    RET

VRESDMC:

END
```

```
$DEBUG

NAME REPETITION_RATE

; subroutine rrate - drives COUNTER0 for 100 ms as a PMT
; pulse repetition rate counter leaving count in bytes
; TH0 and TL0 and overflow condition in bit TF0

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 21, 1985

; link references

    PUBLIC RRATE

    EXTRN CODE (MDELAY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const

    MLCLK  EQU   R4

; var

; begin subroutine

RRATE_CODE SEGMENT CODE
RSEG RRATE_CODE

    RRATE:

            ; clear TIMER0 and overflow FLAG0

           MOV    TL0,#00H        ; lower timer byte
           MOV    TH0,#00H        ; upper  "    "
           CLR    TF0             ; TCON.5 overflow flag
```

```
                ; set TIMER0 to 16 bit counter (MODE 1 counter)
                    ; with interrupt gating disabled while retaining
                ; serial I/O port in auto reload (MODE 3 TIMER1)

            MOV     TMOD,#MOD10+COUNT0+GTOFF0+MOD21

             ; start COUNTER0

            SETB    TR0

                ; start 100ms software count down

            MOV     MLCLK,#100      ; 100 millisec

            RRATE1:

                    CALL    MDELAY
                    DJNZ    MLCLK,RRATE1

            ; stop COUNTER0

            CLR     TR0

             ; return


            RET

        ETARR:

    END
```

```
;$DEBUG

NAME ANALOG_TO_DIGITAL

; subroutine ATOD - drives ADC0816/17 16 channel 8 bit AtoD
; conversion chip including the clearing and setting of the module
; identification strobe to control external memory access preventing
; accidental access to the ATOD peripheral chip. This version is for
; one ADC0816 whose four channel select lines (ADDA .. ADDD)
; are connected to Port 2 (P2.0 .. P2.3). The chip select line is
; P2.7 (ADC).
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 23, 1985

; link  references

        PUBLIC ATOD

        EXTRN CODE (DELAY)

; local  definitions  and  variables
;
; const

        ADC         BIT    P2.7      ; atod chip select
        IOEN        BIT    P3.5      ; input/output enable used
                                     ; here to assert valid PORT2
                                     ; control signals and not
                                     ; upper (external) data memory
; var

     ATOD_DATA      SEGMENT DATA
     RSEG   ATOD_DATA

            CA2D1:      DS     1       ; call addr upper byte
            CA2D2:      DS     1      ;  "    "  lower   "
            ICHAN:      DS     1       ; analog input channel
             IVOLT:      DS     1       ; input voltage

; begin subroutine

ATOD_CODE    SEGMENT CODE
RSEG   ATOD_CODE

     ATOD:

            POP     CA2D2          ; save the
             POP     CA2D1           ;   return address
            POP     ICHAN          ; input channel

                ; disable external data memory protection by asserting
               ; IOEN.  no, absolutely no, external memory references
                ; may be made until this protection is enabled.

             CLR      IOEN            ; disable external data memory
                                     ;  protection

             START:                  ; start atod conversion process

                 MOV     A,ICHAN ; select channel address

                 ORL     A,#11110000B ; mask dtoa strobes

                 MOV     P2,A ; and output channel address

                 CLR     ADC  ; select atod chip

                 CLR     WR   ; latch the address starting conversion
```

```
        CALL   DELAY      ; 12 microsecond delay to allow
        CALL   DELAY      ;   channel address to settle

        SETB   WR         ; turn off write strobe

        SETB   ADC        ; deselect atod chip

    WAIT:               ; wait for interrupt from chip

        JB     IE0,WAIT   ; busy wait

   READ:              ; read the byte from the atod chip

        MOV    P1,#0FFH   ; set up P1 for all input

        ; MOV   A,ICHAN   ; select channel address

        CLR    ADC        ; select atod chip

        CLR    RD         ; turn on read strobe

        CALL   DELAY      ; 12 microsecond delay to allow
        CALL   DELAY      ;   chip output to settle

        MOV    IVOLT,P1   ; read data from input port

        SETB   RD         ; turn off read strobe

        SETB   ADC        ; deselect atod chip
```

```
          DAERR:

              ; enable the external memory access protection

          SETB   IOEN

          PUSH   IVOLT               ; return read voltage

          PUSH   CA2D1               ; load the
          PUSH   CA2D2               ;    return address

          RET

      DOTA:


END
```

```
;$DEBUG

NAME DIGITAL_TO_ANALOG

; subroutine DTOA - drives AD7528 2 channel 8 bit DtoA conversion
; chip including the clearing and setting of the module identification
; strobe to control external memory access preventing accidental
; access to the ATOD and DTOA peripheral chips.
;   SOM version

; author - John F. Babson, University of Hawaii Physics

; revision date - Oct. 20, 1985

; link references

        PUBLIC  DTOA

; local definitions and variables
;
; const

; the following definitions were changed from the original
; values to reflect the "as is" U. of Tokyo hardware
; original values: DAC P2.4, DAE0 P2.5, DAE1 P2.6.

        DAC       BIT     P2.6    ; dtoa !daca/dacb selector
        DAE0      BIT     P2.4    ; AD7528 atod chip #0
        DAE1      BIT     P2.5    ; AD7528 atod chip #1
        IOEN      BIT     P3.5    ; input/output enable used here
                                    ; to assert valid PORT2
                                    ; control signals and not
                                    ; upper (external) data memory
; var

          DTOA_DATA    SEGMENT DATA
          RSEG   DTOA_DATA

                    OCHAN:      DS    1     ; analog output channel
                    OVOLT:      DS    1     ; output voltage
                     CD2A1:     DS    1     ; call address upper byte
                     CD2A2:     DS    1     ;  "    "   lower  "

; ochan contains two bits of significance. bit 0 selects which of two
; separate dtoa chips is selected.  bit 1 determines whether channel
; A or B of an AD7528 dtoa chip is selected.  thus, the system
; contains four dtoa channels.  the information is extracted using
; rotate right with carry then testing the carry bit.

; begin subroutine

DTOA_CODE    SEGMENT CODE
RSEG  DTOA_CODE

          DTOA:

                    POP    CD2A2  ; save the

                    POP    CD2A1  ;   return address

                    POP    OVOLT   ; input the desired output voltage

                    POP    OCHAN   ; input the channel identity

                    MOV    A,OCHAN          ; place chan info in accumulator

                          ; disable the protection from accidental access to PORT2
                          ; based control signals on using external data memory
                          ; by enabling IOEN strobe here.  all code following this
                          ; must make no, repeat no, accesses to external data
                          ; memory (i.e. MOVX) until IOEN is disabled.
```

```
CLR    IOEN

   ; determine AD7528 atod chip to be used

 RRC    A             ; shift bit 0 into (c) flag

 JNC    DTOA1         ; if (c) clr, jump to if seg

       CLR    DAE1          ; else seg - select chip #1

       JMP    DTOA2         ; finish else

 DTOA1:

       CLR    DAE0          ; if seg - select chip #0

 DTOA2:

   ; determine if a or b channel of chip is to be used

 RRC    A             ; shift bit 1 into (c) flag

 JNC    DTOA3         ; if (c) clr, jump to if seq

       CLR    DAC           ; else seq - select channel a

       JMP    DTOA4         ; finish else

 DTOA3:

       SETB   DAC           ; if seq - select channel b

 DTOA4:
        ; having selected the output analog channel, write out
      ; the voltage

    MOV    P1,OVOLT         ; latch analog info.

    CLR    WR               ; start write pulse

    SETB   WR               ; end write pulse

      ; deselect chip

    SETB   DAE0

    SETB   DAE1

        ; deselect channel - simply guarantee state on exciting
      ; this subroutine

    SETB   DAC

        ; finally, disable IOEN to reassert accidental external
      ; data memory access protection.

    SETB   IOEN


    PUSH   CD2A1            ; load the

    PUSH   CD2A2            ;   return address

    RET

AOTD:

END
```

```
$DEBUG

NAME MODULE_INDENTIFICATION
; subroutine MODID - reads the module identification
; hardware indicating the binary identification of a STRING
; module (instrument)

; author - John F. Babson, University of Hawaii Physics

; revision date - Oct. 20, 1985

; link references

    PUBLIC MODID

; local definitions and variables
;
; const

    IOEN        BIT    P3.5   ; I/O enable
    MIDS        BIT     P3.3    ; identity circuit chip select


; var

    MODID_DATA     SEGMENT DATA
    RSEG  MODID_DATA

         CA2D1:      DS     1       ; call addr upper byte
         CA2D2:      DS     1      ;  "    "  lower   "
         MID:        DS     1       ; module identity
```

```
; begin subroutine

MODID_CODE   SEGMENT CODE
RSEG   MODID_CODE

     MODID:

            POP     CA2D2          ; save the
            POP     CA2D1          ;   return address


              MODID1:              ; read the byte from the identity circuit

                 MOV    P1,#0FFH        ; set up P1 for all input

                 CLR    IOEN              ; enable I/O operation
                 CLR    MIDS          ; select ident circuit

                 MOV    A,P1          ; read data from input

                 SETB   MIDS          ; deselect ident circuit
                 SETB   IOEN          ; disable I/O operation

                 ANL    A,#00001111B    ; mask out unused 4 msb's

                 MOV    MID,A         ; and save binary identity
                                      ;   for return

           MODID2:

           PUSH  MID                  ; return module identity

           PUSH  CA2D1                ; load the
           PUSH  CA2D2                ;   return address

           RET

     DIDOM:

END
```

```
; $DEBUG

NAME SET_DESCRIMINATOR_THRESHOLD_LEVEL

; subroutine SETDSCR - used to set the discriminator threshold level
; of the PMT
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

; link references
;
     PUBLIC SETDSCR

     EXTRN CODE (HEXBIN, DTOA, UPDATE, CREPLY)

; global variables

     EXTRN DATA (CMD, DATUM)

$INCLUDE(CONSNT.INC)


; local definitions and variables
;
; const

; var

DISCR_DATA    SEGMENT DATA
RSEG   DISCR_DATA

     DSCMD:      DS     1      ; discriminator command
     THRESH:     DS     1      ; binary value of threshold level
     DUMMY:      DS     1      ; dummy cell
```

```
; begin subroutine

DISCR_CODE   SEGMENT CODE
RSEG    DISCR_CODE

    SETDSCR:

                ; fetch the descriminator channel

          MOV     DUMMY,#NUM0

          PUSH    DUMMY

          PUSH    CMD+1

          CALL    HEXBIN

          POP     DSCMD

                ; fetch the desired value of threshold level

          PUSH    DATUM

          PUSH    DATUM+1

          CALL    HEXBIN

          POP     THRESH

                ; write it to the pmt discriminator

          PUSH    DSCMD           ; discriminator channel

          PUSH    THRESH

          CALL    DTOA
```

```
                ; update the analog table reading actual level

            PUSH   DSCMD

            CALL   UPDATE

                 ; finish reply to command returning actual
               ; threshold level

            CALL   CREPLY

            RET

      RCSDTES:

END
```

```
; $DEBUG

NAME SET_HIGH_VOLTAGE

; subroutine SETHV - used to set the high voltage level of the PMT
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

; link references
;
      PUBLIC SETHV

      EXTRN CODE (HEXBIN, ANALKUP, RAMPUP, RAMPDN, UPDATE,
                  CREPLY)
      EXTRN CODE (XGTY)

; global variables

      EXTRN DATA (CMD, DATUM)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const

; var

SETHV_DATA    SEGMENT DATA
RSEG   SETHV_DATA

      HVCMD:      DS     1      ; hi volt command
      OLDV:       DS     1      ; old volt value
      NEWV:       DS     1      ; new volt value
      DUMMY:      DS     1      ; dummy cell
```

```
; begin subroutine

SETHV_CODE    SEGMENT CODE
RSEG   SETHV_CODE

     SETHV:

               ; lookup the current PMT high voltage setting

          MOV    DUMMY,#NUM0       ; call hexbin(zero,cmd2,hvcmd)

          PUSH   DUMMY

          PUSH   CMD+1

          CALL   HEXBIN

          POP    HVCMD

          PUSH   HVCMD         ; call analkup(hvcmd,oldv)

          CALL   ANALKUP

          POP    OLDV

               ; fetch the desired PMT high voltage value from the
             ; command buffer

          PUSH   DATUM          ; call hexbin(datum1,datum2,newv)

          PUSH   DATUM+1

          CALL   HEXBIN

          POP    NEWV
```

```
        ; ramp up or down the PMT high voltage from the current
        ; setting accordingly

       ; if (newv > oldv) then

  %MIF(NEWV,MXGTY,OLDV,SETHV1)

           PUSH   NEWV   ; call rampup(newv, volt)

         PUSH   OLDV

         CALL   RAMPUP

  %MELSE(SETHV1,SETHV2)

           PUSH   NEWV   ; call rampdn(newv, volt)

         PUSH   OLDV

         CALL   RAMPDN

   SETHV2:

        ; update the analog table reading actual voltage setting

  PUSH   HVCMD

  CALL   UPDATE

        ; finish reply to command returning actual threshold level

  CALL   CREPLY

  RET

VHTES:
END
```

```
$DEBUG

NAME COMMAND_ERROR

; subroutine CMDERR - used to respond to command error condition by
; echoing back improper command with a question mark
;   STRING module version

; message string:
;    <$><DEV><DEV><CMD(0)><CMD(1)><?><?><CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - May 20, 1986

;
; link references
;
      PUBLIC CMDERR
      EXTRN CODE (PUT_CHAR, XEQY)

; global variables

      EXTRN DATA (DEV, CMD, OUTCF)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const

; var

; begin subroutine

CMDERR_CODE  SEGMENT CODE
RSEG    CMDERR_CODE


      CMDERR:

                  ; if (multiple devices called) {
```

```
        %MIF (OUTCF,MXEQY,#FALSE,CMDERR1)

                RET     ; i.e. send back no messages

          ; }

         CMDERR1:

           ; echo the erroroneous command

          MOV     A,#DOLLAR         ; synchronization  byte
          CALL    PUT_CHAR

          MOV     A,DEV             ; device  byte
          CALL    PUT_CHAR          ;   echoed
          CALL    PUT_CHAR          ;      twice

          MOV     A,CMD             ; first  command  byte  echoed
          CALL    PUT_CHAR

          MOV     A,CMD+1           ; second command byte echoed
          CALL    PUT_CHAR

          MOV     A,#QUESTION       ; question  mark  twice
          CALL    PUT_CHAR
          CALL    PUT_CHAR

          MOV     A,#CR
          CALL    PUT_CHAR          ;<CR>


          RET

     RREDMC:

END
```

```
;$DEBUG

NAME DEVICE_ERROR

; subroutine DEVICE -used to respond to device error condition by
; sending back error message with a question mark
;   STRING optical module
;
; message string:
;
;     <$>DEVICE?<CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 13, 1985

;
; link references
;

     PUBLIC DEVERR

     EXTRN CODE (PUT_STRING, PUT_CHAR)

; global variables

$INCLUDE(CONSNT.INC)


; local definitions and variables
;
; const

DEVERR_MSG    SEGMENT CODE
RSEG   DEVERR_MSG

       DEVMSG_1:      DB      '$DEVICE?',00H  ; device error message

; var
; begin  subroutine

DEVERR_CODE   SEGMENT CODE
RSEG    DEVERR_CODE


      DEVERR:

                ; send the device error message

              MOV     DPTR,#DEVMSG_1
              CALL    PUT_STRING

              MOV     A,#CR
              CALL    PUT_CHAR

              RET

      RREVED:

END
```

```
;$DEBUG

NAME FORMAT_ERROR

; subroutine FORERR -used to respond to message format
; error condition by sending back error message "WHAT"
; indicating format error caught by addressed device.  Usually
; this message will be ignored by the SBC microcomputer since
; most likely the format error will be caught by that computer.
; If this message gets through, it is an indication of communi-
; cations problems between the SBC microcomputer and the
; remote device. This error condition serves the additional
; purpose of testing individual devices in the laboratory.
;   STRING optical module / POWER module version
;
; message string:
;
;    <$><DEV><DEV>WHAT<CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - June 30, 1986

;
; link references
;

    PUBLIC FORERR

    EXTRN CODE (PUT_STRING, PUT_CHAR, XEQY)

; global variables

    EXTRN DATA (DEV, OUTCF)
```

```
$INCLUDE(UHPS.INC)


; local definitions and variables
;
; const

FORERR_MSG    SEGMENT CODE
RSEG  FORERR_MSG

        FORMSG_1:       DB      'WHAT',00H  ; format error message

; var

; begin subroutine

FORERR_CODE   SEGMENT CODE
RSEG    FORERR_CODE


      FORERR:

                ; if (multiple devices called) {

           %MIF (OUTCF,MXEQY,#FALSE,FORERR1)

                    RET     ; i.e. send back no messages

             ; }

            FORERR1:

             MOV    A,#DOLLAR          ; synchronization  byte
             CALL   PUT_CHAR

             MOV    A,DEV              ; device  byte
             CALL   PUT_CHAR           ;   echoed
             CALL   PUT_CHAR           ;      twice
        ; send the format error message

      MOV    DPTR,#FORMSG_1
      CALL   PUT_STRING

      MOV    A,#CR
      CALL   PUT_CHAR

      RET

   RREROF:

END
```

```
$DEBUG

NAME IS_IT_ALPHA_NUMERIC

; SUBROUTINE ISALNO
;
; routine to check whether or not a candidate character
; is a valid ASCII alphanumeric value or not
;
; link references
;

      PUBLIC  ISALNO
     EXTRN CODE (XGEY, XLEY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const

; var

ISALNO_DATA SEGMENT DATA
RSEG ISALNO_DATA

      CISANO1:    DS    1       ; return
      CISANO2:    DS    1      ;   address
      CHAR:       DS    1       ; character buffer
      EFLAG:      DS    1       ; condition flag
```

```
; begin subroutine

ISALNO_CODE SEGMENT CODE
RSEG ISALNO_CODE

      ISALNO:

            POP    CISANO1    ; save return
            POP    CISANO2    ;   address

            POP    CHAR       ; the candidate alphanumeric character

              ; set error flag EFLAG = #FALSE
                ; this is the default condition for
               ; the following comparison

            MOV    EFLAG,#FALSE

              ; now, successively compare CHAR to
                ; see if it is a valid ASCII alphanumeric character
              ; setting EFLAG = #TRUE iff it is

              ; case - is it a number, then set eflag TRUE

           %MIF (CHAR,MXGEY,#NUM0,ISALNO1)

                %MIF (CHAR,MXLEY,#NUM9,COMPEND)

                        MOV    EFLAG,#TRUE

            ISALNO1:

                ; case - is it a capital letter, then set eflag TRUE

           %MIF (CHAR,MXGEY,#BIGA,ISALNO2)

                %MIF (CHAR,MXLEY,#BIGZ,COMPEND)

                        MOV    EFLAG,#TRUE
```

```
    ISALNO2:

        ; case - is it a small letter, then set eflag TRUE

    %MIF(CHAR,MXGEY,#LETA,ISALNO3)

        %MIF (CHAR,MXLEY,#LETZ,COMPEND)

                MOV    EFLAG,#TRUE

    COMPEND:        ; end comparisons

      ; now, return

    PUSH   EFLAG      ; return  condition

    PUSH   CISANO2  ; restore  return
    PUSH   CISANO1  ;   address

    RET               ; return

  ONLASI:

END
```

```
$DEBUG

NAME IS_IT_HEX

; SUBROUTINE ISHEX
;
; routine to check whether or not a candidate character
; is a valid ASCII hex value or not
;
; link references
;

      PUBLIC  ISHEX
     EXTRN CODE (XGEY, XLEY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const

; var

ISHEX_DATA SEGMENT DATA
RSEG ISHEX_DATA

      CISHEX1:    DS     1      ; return
      CISHEX2:    DS     1      ;  address
      CHAR:       DS     1      ; character buffer
      EFLAG:      DS     1      ; condition flag
```

```
; begin subroutine

ISHEX_CODE SEGMENT CODE
RSEG ISHEX_CODE

    ISHEX:

          POP    CISHEX1    ; save return
          POP    CISHEX2    ;   address

          POP    CHAR       ; the candidate hex character

            ; set error flag EFLAG = #FALSE
              ; this is the default condition for
             ; the following comparison

          MOV    EFLAG,#FALSE

            ; now, successively compare CHAR to
             ; see if it is a valid ASCII hex character
            ; setting EFLAG = #TRUE iff it is

        %MIF (CHAR,MXGEY,#NUM0,COMPEND)

            %MIF (CHAR,MXLEY,#BIGF,COMPEND)

                %MIF (CHAR,MXLEY,#NUM9,ISHEX1)

                    MOV    EFLAG,#TRUE

                %MELSE (ISHEX1,COMPEND)

                    %MIF(CHAR,MXGEY,#BIGA,COMPEND)

                        MOV    EFLAG,#TRUE

          COMPEND:       ; end comparisons
          ; now, return

         PUSH   EFLAG      ; return condition

         PUSH   CISHEX2    ; restore return
         PUSH   CISHEX1    ;  address

         RET               ; return

     XEHSI:

END
```

```
;$DEBUG

NAME RAMP_DOWN_PMT_VOLTAGE

; subroutine RAMPDN - ramps down PMT voltage - set HIVOLT channel
; to reflect actual  hardware
;  SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 22, 1985

; link references
;
     PUBLIC RAMPDN

     EXTRN CODE (DTOA, HEXBIN, MDELAY)
     EXTRN CODE (XGTY)

; global variables

     EXTRN DATA (CMD)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const

     DELTIME     EQU    19H         ; 25 millisec delay
     MLCLK       EQU    R5          ; millisecond clock
```

```
; var

    RAMPDN_DATA     SEGMENT DATA
    RSEG   RAMPDN_DATA

        RANGDN:     DS     1     ; new voltage (target)
        VOLTDN:     DS     1     ; running voltage, initially old
        DTOACH:     DS     1     ; d to a channel parameter
        CRMDN1:     DS     1     ; save address upper byte
        CRMDN2:     DS     1     ;  "     "   lower   "
        DUMMY:      DS     1     ; dummy cell

; begin  subroutine

RAMPDN_CODE     SEGMENT CODE
RSEG   RAMPDN_CODE

    RAMPDN:

        POP     CRMDN2  ; save the
        POP     CRMDN1  ;   return address

        POP     VOLTDN  ; input current voltage
        POP     RANGDN  ; input target voltage

          ; load the high voltage channel identity into the d to a
        ; channel

       MOV     DUMMY,#NUM0

       PUSH    DUMMY

       PUSH    CMD+1

       CALL    HEXBIN

       POP     DTOACH
```

```
          ; while (voltdn > rangdn)

        %MWHILE(VOLTDN,MXGTY,RANGDN,RDN1,RDN2)

              ; voltdn := voltdn -1

             DEC    VOLTDN        ; binary subtraction

             ; place voltage change on pmt

           PUSH   DTOACH

           PUSH   VOLTDN

           CALL   DTOA

             ; delay one step time

           MOV    MLCLK,#DELTIME

           CALL   MDELAY

        %MWEND(RDN1,RDN2)

         PUSH    CRMDN1  ; load the
         PUSH    CRMDN2  ;   return address

         RET

     NDPMAR:

END
```

```
;$DEBUG

NAME RAMP_UP_PMT_VOLTAGE

; subroutine RAMPUP - ramps up the PMT voltage - set
; HIVOLT channel to reflect actual hardware
;   SOM version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 22, 1985

; link references
;
      PUBLIC RAMPUP

      EXTRN CODE (DTOA, HEXBIN, MDELAY)
      EXTRN CODE (XGTY)

; global variables

      EXTRN DATA (CMD)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const

      DELTIME     EQU   19H       ; 25 millisec delay
      MLCLK       EQU   R5        ; milliseconf clock
```

```
; var

     RAMPUP_DATA     SEGMENT DATA
     RSEG   RAMPUP_DATA

          RANGUP:     DS     1     ; new voltage (target)
          VOLTUP:     DS     1     ; running voltage, initially old
          DTOACH:     DS     1     ; d to a channel parameter
          CRMUP1:     DS     1     ; save address upper byte
          CRMUP2:     DS     1     ;  "      "   lower  "
          DUMMY:      DS     1     ; dummy cell

; begin  subroutine

RAMPUP_CODE     SEGMENT CODE
RSEG   RAMPUP_CODE

     RAMPUP:

          POP     CRMUP2  ; save the
          POP     CRMUP1  ;   return address

          POP     VOLTUP  ; input current voltage
          POP     RANGUP  ; input target voltage

             ; load the high voltage channel identity into the d to a
           ; channel

          MOV     DUMMY,#NUM0

          PUSH    DUMMY

          PUSH    CMD+1

          CALL    HEXBIN

          POP     DTOACH
```

```
                ; while (rangup > voltup)

            %MWHILE(RANGUP,MXGTY,VOLTUP,RUP1,RUP2)

                      ; voltup := voltup +1

                    INC      VOLTUP          ; binary addition

                      ; place voltage change on pmt

                    PUSH     DTOACH

                    PUSH     VOLTUP

                    CALL     DTOA

                      ; delay one step time

                    MOV      MLCLK,#DELTIME

                    CALL     MDELAY

            %MWEND(RUP1,RUP2)

              PUSH    CRMUP1  ; load the
              PUSH    CRMUP2  ;   return address

              RET

        PUPMAR:

    END
```

APPENDIX E

Source Code Listing of the Executive Program for Controlling the Power Module (PWR)

The source code for program PWR consists of 10 modules which are listed below roughly in a top down fashion with the highest level routines being listed first and the lowest level or direct hardware driving modules listed last. This pattern approximates the order in which modules are called in the program and is listed according to the order shown in Table E.1 below:

Table E.1 Listing Order of PWR Routines

Class	Name	Description
Main:		
	PWR	Main program
	INITIAL	Initialize stack and 300 baud serial port
Communications control:		
	ECHOM	Echo back valid command
	SYNCH	Synchronize on first byte of command string
Parser:		
	CMDPOLL	Parse the command string for a valid command
Command executive:		
	CMDSERV	Service the command request

Device drivers:

PWRON	Power on specified device
PWROFF	Power off specified device

Error handling:

CMDERR	Report a command error
FORERR	Report a format error

The listing of the individual modules now follows:

```
$DEBUG

NAME POWER_MODULE

; program POWER
;
; program is first attempt to control the DUMAND POWER
; module. this is a very elementry program taking all control
; directly from a 300 baud terminal sending only printable
; ASCII characters.

; author - John F. Babson, University of Hawaii Physics
; revision date - May 12, 1986

;
; link references
;
    EXTRN CODE (INITIAL, SYNCH, CMDPOLL, CMDSERV)
    EXTRN CODE (XEQY, XNEY, XGEY, XGTY)
    EXTRN CODE (PWRON, PWROFF)

$INCLUDE(UHPS.INC)


; global definitions and variables
;
; const

    MSCOUNT    EQU    R5     ; ms counter
```

```
; var

GLOBAL_DATA     SEGMENT DATA
RSEG   GLOBAL_DATA

     DEV:          DS      1        ; device register (hex)
     CMD:          DS      2        ; command buffer (hex)
     DATUM:        DS      2        ; data buffer (hex)
     ; ANALOG:     DS      MAXCHAN ; analog look up table (binary)
     ON:           DS       2         ; power module switch status buffer
     ERRFLG:       DS       1        ; communications error flag
     SYNCFLG:      DS      1        ; communications
                                   ;    synchronization  flag

     PUBLIC DEV, CMD, DATUM, ON, ERRFLG, SYNCFLG
     ; PUBLIC ANALOG

; local definitions and variables

; const

     DEVICE        EQU   CMD+1    ; identity of device to be turned on or
                                 ; off
     LATCH0        BIT   P3.3
     LATCH1        BIT   P3.4

; var

POWER_DATA      SEGMENT DATA
RSEG          POWER_DATA

     LOOP:  DS     1          ; infinite loop condition (binary)
```

```
; begin program

; POWER_CODE    SEGMENT CODE
; RSEG   POWER_CODE

CSEG

ORG      0          ; system start up location

POWER:

        CALL    INITIAL         ; initialize system

         ; note: in the following some non-existent devices may be
        ; "turned on or off" but that is OKAY

         ; turn off all of the devices

         ; point to first hex numbered device

      MOV    DEVICE,#NUM0

       ; repeat {

     %MREPEAT (POWER1)

           CALL   PWROFF          ; turn off device

           INC    DEVICE

         ; } until (device(0) == #NUM9+1) /* last numbered device */

     %MUNTIL (DEVICE,MXGTY,#NUM9,POWER1,POWER2)

         ; point to first hex lettered device

      MOV    DEVICE,#BIGA
```

```
  ; repeat {

%MREPEAT (POWER3)

        CALL   PWROFF

        INC    DEVICE

   ; } until (device(0) > #BIGB) /* last lettered device used */

%MUNTIL (DEVICE,MXGTY,#BIGB,POWER3,POWER4)

   ; doubly ensure that all devices are turned off

 MOV    ON,#0FFH
 MOV    P1,ON

 SETB   LATCH0
 NOP
 NOP
 CLR    LATCH0

 MOV    ON+1,#0FFH
 MOV    P1,ON+1

 SETB   LATCH1
 NOP
 NOP
 CLR    LATCH1

  ; loop := true

MOV    LOOP,#TRUE
```

```
      ; while (loop = true)
    %MWHILE(LOOP,MXEQY,#TRUE,POWER9,POWER10)
            ; loop := true
          MOV    LOOP,#TRUE
             ; read command line and parse it for command
         %MREPEAT (ERROR1)
                    ; loop for first synch byte
               CALL   SYNCH
              %MREPEAT (ERROR3)
                          ; loop for the remaining bytes
                     CALL   CMDPOLL
             %MUNTIL (SYNCFLG,MXNEY,#TRUE,ERROR3,ERROR4)
        %MUNTIL (ERRFLG,MXNEY,#TRUE,ERROR1,ERROR2)
            ; poll all analog devices
          ; CALL   ANAPOLL
            ; service the command
          CALL   CMDSERV
      ; end while
    %MWEND(POWER9,POWER10)
REWOP:
END
```

```
NAME INITIAL

; subroutine INITIAL - used to initialize the system hardware
; including the serial I/O port, the system stack, and any
; AtoD and DtoA hardware.
;   UNIVERSAL 300 baud version

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 15, 1985

;
; link references
;

        PUBLIC  INITIAL

; global definitions and variables
;
; const

$INCLUDE (CONSNT.INC)


; local definitions and variables
;
; const

         ; control of serial port

        BAU300      EQU 99H         ; 300 baud timer (self determined)
        SERPORT     EQU 11011010B   ; Serial Port
```

```
; var

DSEG at    8

STACK:     DS    24    ; at power-up, the stack pointer is
                       ; initialized to point here.
                       ; 24 byte space reserved.

; begin subroutine

INITIAL_CODE    SEGMENT CODE
RSEG INITIAL_CODE

    INITIAL:

            ; This is the initializing section. Execution always
          ; starts at address 0 on power-up.

          MOV    TMOD,#MOD21      ; set timer1 mode to auto-reload
          MOV    TH1,#BAU300     ; set timer for 300 BAUD
       MOV    SCON,#SERPORT  ; prepare the Serial Port
        SETB   TR1              ; start clock

        RET

    LAITINI:

END
```

```
$DEBUG

NAME ECHO_MESSAGE

; subroutine ECHOM - used to echo back write only commands
;   POWER version - uses OUT_CHAR

; message string:
;     <CMD(0)><CMD(1)><CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - May 12, 1986

;
; link references
;

      PUBLIC ECHOM

      EXTRN CODE (OUT_CHAR)

; global variables

      EXTRN DATA (CMD)

$INCLUDE(CONSNT.INC)


; local definitions and variables
;
; const

; var
; begin  subroutine

ECHOM_CODE   SEGMENT CODE
RSEG    ECHOM_CODE


      ECHOM:

             ; echo the command

              MOV     A,CMD          ; first command byte echoed
             CALL   OUT_CHAR

              MOV     A,CMD+1       ; second command byte echoed
             CALL   OUT_CHAR

             MOV     A,#CR         ; <CR>
             CALL   OUT_CHAR

             RET

      MOHCE:

END
```

```
;$DEBUG

NAME SYNCHRONIZE

; subroutine SYNCH - parses for initial synchronization byte <$>
; in input data stream - any additional synchronization bytes
; are handled in CMDPOL
;   STRING / POWER module version

; author - John F. Babson, University of Hawaii Physics
; revision date - Apr. 30, 1986

;
; link references
;

      PUBLIC SYNCH

      EXTRN CODE (GET_CHAR, XEQY)

$INCLUDE(UHPS.INC)


; local definitions and variables
;
; const

; var

SYNCH_DATA    SEGMENT DATA
RSEG     SYNCH_DATA

        SYNBYTE:      DS        1  ; synchronization byte
```

```
; begin subroutine

SYNCH_CODE   SEGMENT CODE
RSEG     SYNCH_CODE


     SYNCH:

               ; loop until synch byte <$> found

          %MREPEAT (SYNCH1)

               CALL    GET_CHAR

               MOV     SYNBYTE,A

          %MUNTIL (SYNBYTE,MXEQY, #DOLLAR, SYNCH1, SYNCH2)

           RET

     HCNYS:

END
```

```
$DEBUG

NAME COMMAND_POLL

; routine used to poll for a valid command
;   POWER module version

; author - John F. Babson, University of Hawaii Physics
; revision date - May 13, 1986

; note: unbuffered communications!  each character fetched by
; "CALL GET_CHAR" rather than read out of a string buffer.

;
; link references
;

     PUBLIC CMDPOLL

     EXTRN CODE (GET_CHAR, OUT_CHAR, PUT_STRING, PUT_CRLF,
                 XEQY, XNEY)
     EXTRN CODE (ASCBIN)

; global variables

     EXTRN DATA (DEV, CMD, DATUM, ERRFLG, SYNCFLG)

$INCLUDE(UHPS.INC)

; local definitions and variables
;
; const
; var

CMDPOL_DATA   SEGMENT DATA
RSEG    CMDPOL_DATA

        CHAR:          DS     1       ; character buffer
        DEVID:         DS     1       ; binary form of DEV

; macro definitions

%*DEFINE (MSYNCH(LAB)) LOCAL LABEL

(

      CALL    GET_CHAR

      MOV     CHAR,A

      %MIF (CHAR,MXEQY,#DOLLAR,SYN%LAB)

            MOV     SYNCFLG,#TRUE

             RET

      SYN%LAB:

)


; begin subroutine

CMDPOL_CODE  SEGMENT CODE
RSEG   CMDPOL_CODE

CMDPOLL:

          ; initialize control flags

         ; disable error flag

      MOV     ERRFLG,#FALSE
```

```
  ; disable message synch flag
 MOV    SYNCFLG,#FALSE


    ; first check for consistent device destination
  %MSYNCH(1)       ; first device byte in DEV
 MOV    DEV,CHAR
  %MSYNCH(2)      ; second device byte in CHAR
%MIF (DEV,MXNEY,CHAR,CMDPOL1)
      MOV    ERRFLG,#TRUE    ; inconsistent device
                             ; so look for next message
       RET
CMDPOL1:


   ; second check if message is for this module
  PUSH    DEV      ; convert DEV info to binary
  CALL    ASCBIN  ; form in order to compare
  POP     DEVID   ; it to the device identity
  ; if (devid != 0XA) {
%MIF (DEVID,MXNEY,#0AH,CMDPOL9)
     MOV    ERRFLG,#TRUE     ; not for POWER
                             ;   module so return
      RET
```

```
%MELSE(CMDPOL9,CMDPOL9A)

          ; message unique for this module so echo back the
          ; header for this command

       MOV    A,#DOLLAR
       CALL   OUT_CHAR

       MOV    A,#BIGA
       CALL   OUT_CHAR
       CALL   OUT_CHAR

CMDPOL9A:

   ; third parse message for a command

  %MSYNCH(3)   ; first command byte

MOV    CMD,CHAR

 %MSYNCH(4)   ; second command byte

MOV    CMD+1,CHAR

  %MSYNCH(5)   ; termination or first data byte

MOV    DATUM,CHAR

 ; if (datum == <CR>) {

%MIF (DATUM,MXEQY,#CR,CMDPOL10)

        RET     ; message  completed

CMDPOL10:

 %MSYNCH(6)   ; second data byte

MOV    DATUM+1,CHAR
```

```
    ; check for command termination
  %MSYNCH(7)   ; command terminator in CHAR
  ; if (char == <CR>) {
%MIF (CHAR,MXEQY,#CR,CMDPOL11)
        RET     ; message completed
CMDPOL11:
    ; message not properly terminated
MOV    ERRFLG,#TRUE
RET
LLOPDMC:
END
```

```
$DEBUG

NAME COMMAND_SERVICE

; routine services the command or returns an error message
; for invalid commands.  it is here that the various output
; channels are defined.
;   POWER module version.

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 14, 1985

; link references

        PUBLIC CMDSERV

        EXTRN CODE (XEQY, CMDERR, PWROFF, PWRON, ECHOM)

$INCLUDE(UHPS.INC)

; global constants and variables

        EXTRN DATA (CMD)

; local definitions and variables
;
; const

; var
; begin  subroutine

CMD_SERV_CODE      SEGMENT CODE
RSEG    CMD_SERV_CODE

CMDSERV:

        ; if (cmd(0) = 'S') then

     %MIF(CMD,MXEQY,#BIGS,CSERV1)

            CALL    PWROFF
             CALL    ECHOM   ; echo command message

     %MELSE(CSERV1,CSERV2)

              ; if (cmd(0) = 'Q') then

           %MIF(CMD,MXEQY,#BIGQ,CSERV5)

                  CALL    PWRON
                   CALL    ECHOM   ; echo command message

           %MELSE(CSERV5,CSERV6)

                  CALL    CMDERR ; echo back bad command

            CSERV6:

     CSERV2:

     RET

VRESDMC:

END
```

```
$DEBUG

NAME POWER_ON

; subroutine pwron - turns on the requested device.  Only one
; device maybe turned on at a time.  The global data structure
; ON(2) is used with "ones" indicating the associated device is
; on and "zeros" indicating off.  cmd(1) is used to contain the
;  device  identification.

; author - John F. Babson, University of Hawaii Physics
; revision date - Nov. 7, 1985

; calling  sequence:
;  EX:  CALL    PWRON   ; device information is hex number stored
;                       ;   in global byte cmd(1)

;  link  references

      PUBLIC PWRON
      EXTRN CODE (CMDERR, ISHEX, XEQY, XLEY)

; global  constants  and  variables
;
; const

$INCLUDE (UHPS.INC)

; var

      EXTRN DATA (CMD, ON)

;  local  definitions  and  variables
;
; const

       LATCH0  BIT    P3.3
       LATCH1  BIT    P3.4
```

```
; var

PWRON_DATA SEGMENT DATA
RSEG PWRON_DATA

        RESULT:        DS     1        ; result of hex test

; begin  subroutine

PWRON_CODE SEGMENT CODE
RSEG PWRON_CODE

      PWRON:

                    ; check if device identifier is valid hex identifier

            PUSH    CMD+1   ; pass the device identity

           CALL   ISHEX

           POP     RESULT

          %MIF (RESULT,MXEQY,#FALSE,PWRON1)

                         ; erroneous device, exit with error message

                CALL   CMDERR

                 RET

           PWRON1:
```

```
        ; next, check if device controlled by LATCH0 [0..07H]
       ; or LATCH1 [08H..0BH] and set appropriate bit by
         ; picking if off and ORing it with the existing bits.

%MIF (CMD+1,MXLEY,#NUM7,PWRON2)

            ; LATCH0 cases - compare cmd(1) to 0..7H

            ; switch (cmd(1))

        MOV    A,CMD+1

         ; case (00H)

        CJNE   A,#NUM0,PWRON20

                ANL    ON,#11111110B
              JMP    PWRON27

        PWRON20:

         ; case (01H)

        CJNE   A,#NUM1,PWRON21

                ANL    ON,#11111101B
              JMP    PWRON27

        PWRON21:

         ; case (02H)

        CJNE   A,#NUM2,PWRON22

                ANL    ON,#11111011B
              JMP    PWRON27

        PWRON22:
```

```
 ; case (03H)

CJNE    A,#NUM3,PWRON23

        ANL     ON,#11110111B
      JMP     PWRON27

PWRON23:

 ; case (04H)

CJNE    A,#NUM4,PWRON24

        ANL     ON,#11101111B
      JMP     PWRON27

PWRON24:

 ; case (05H)

CJNE    A,#NUM5,PWRON25

        ANL     ON,#11011111B
      JMP     PWRON27

PWRON25:

 ; case (06H)

CJNE    A,#NUM6,PWRON26

        ANL     ON,#10111111B
      JMP     PWRON27
```

```
PWRON26:

  ; default

         ANL    ON,#01111111B

 PWRON27:       ; end case

    ; move result to LATCH0 and write it out

  MOV     P1,ON   ; place it on port

  SETB   LATCH0  ; write
NOP
NOP
 CLR    LATCH0  ;   it out

%MELSE(PWRON2,PWRON3)

   ; latch1 cases - compare cmd(1) to 8H..0BH

   ; switch (cmd(1))

MOV    A,CMD+1

 ; case (08H)

CJNE   A,#NUM8,PWRON28

         ANL    ON+1,#11111110B
      JMP    PWRON2C

PWRON28:

 ; case (09H)

CJNE   A,#NUM9,PWRON29

         ANL    ON+1,#11111101B
      JMP    PWRON2C
```

```
PWRON29:

 ; case (0AH)

CJNE   A,#BIGA,PWRON2A

        ANL    ON+1,#11111011B
      JMP    PWRON2C

PWRON2A:

 ; case (0BH)

CJNE   A,#BIGB,PWRON2B

        ANL    ON+1,#11110111B
      JMP    PWRON2C

PWRON2B:

  ; default case
      ; if it falls through, to have non-existent
  ; device selected

       CALL    CMDERR  ; "devices" C thru F

PWRON2C:

     ; move result to LATCH1 and write it out

 MOV     P1,ON+1         ; place it on port

 SETB    LATCH1          ; write
NOP
NOP
 CLR     LATCH1          ;   it out
```

```
          PWRON3:

            ; return

            RET

      NORWP:

END
```

```
$DEBUG

NAME POWER_OFF

; subroutine pwroff - turns on the requested device.  Only one
; device maybe turned on at a time.  The global data structure
; ON(2) is used with "ones" indicating the associated device is
; on and "zeros" indicating off.  cmd(1) is used to contain the
;  device  identification.

; author - John F. Babson, University of Hawaii Physics
; revision date - Nov. 7, 1985

; calling sequence:
;  EX:  CALL    PWROFF   ; device information is hex number stored
;                        ;  in global byte cmd(1)

; link references

    PUBLIC PWROFF
    EXTRN CODE (CMDERR, ISHEX, XEQY, XLEY)

; global constants and variables
;
; const

$INCLUDE (UHPS.INC)

; var

    EXTRN DATA (CMD, ON)

; local definitions and variables
;
; const

     LATCH0  BIT    P3.3
     LATCH1  BIT    P3.4
```

```
; var

PWROFF_DATA SEGMENT DATA
RSEG PWROFF_DATA

        RESULT:        DS      1        ; result of hex test

; begin  subroutine

PWROFF_CODE SEGMENT CODE
RSEG PWROFF_CODE

      PWROFF:

                    ; check if device identifier is valid hex identifier

                PUSH    CMD+1   ; pass the device identity

              CALL   ISHEX

              POP     RESULT

            %MIF (RESULT,MXEQY,#FALSE,PWROFF1)

                         ; erroneous device, exit with error message

                  CALL   CMDERR

                   RET

              PWROFF1:
```

```
      ; next, check if device controlled by LATCH0 [0..07H]
     ; or LATCH1 [08H..0BH] and set appropriate bit by
       ; picking if off and ANDing it's compliment with
     ; the existing bits.

%MIF (CMD+1,MXLEY,#NUM7,PWROFF2)

           ; LATCH0 cases - compare cmd(1) to 0..7H

           ; switch (cmd(1))

        MOV    A,CMD+1

          ; case (00H)

        CJNE   A,#NUM0,PWROFF20

             ORL    ON,#00000001B
            JMP    PWROFF27

        PWROFF20:

          ; case (01H)

        CJNE   A,#NUM1,PWROFF21

             ORL    ON,#00000010B
            JMP    PWROFF27

        PWROFF21:

          ; case (02H)

        CJNE   A,#NUM2,PWROFF22

             ORL    ON,#00000100B
            JMP    PWROFF27

        PWROFF22:
```

```
 ; case (03H)

CJNE    A,#NUM3,PWROFF23

          ORL     ON,#00001000B
        JMP     PWROFF27

PWROFF23:

 ; case (04H)

CJNE    A,#NUM4,PWROFF24

          ORL     ON,#00010000B
        JMP     PWROFF27

PWROFF24:

 ; case (05H)

CJNE    A,#NUM5,PWROFF25

          ORL     ON,#00100000B
        JMP     PWROFF27

PWROFF25:

 ; case (06H)

CJNE    A,#NUM6,PWROFF26

          ORL     ON,#01000000B
        JMP     PWROFF27

PWROFF26:

 ; default

          ORL     ON,#10000000B

PWROFF27:       ; end case
```

```
        ; move result to LATCH0 and write it out

      MOV     P1,ON   ; place it on port

      SETB    LATCH0  ; write
     NOP
     NOP
      CLR     LATCH0  ;   it out

%MELSE(PWROFF2,PWROFF3)

       ; switch (cmd(1))

     MOV     A,CMD+1

      ; case (08H)

     CJNE    A,#NUM8,PWROFF28

             ORL     ON+1,#00000001B
           JMP     PWROFF2C

     PWROFF28:

      ; case (09H)

     CJNE    A,#NUM9,PWROFF29

             ORL     ON+1,#00000010B
           JMP     PWROFF2C

     PWROFF29:
        ; case (0AH)

        CJNE    A,#BIGA,PWROFF2A

                ORL     ON+1,#00000100B
              JMP     PWROFF2C

        PWROFF2A:

        ; case (0BH)

        CJNE    A,#BIGB,PWROFF2B

                ORL     ON+1,#00001000B
              JMP     PWROFF2C

         PWROFF2B:  ; default case
                      ; if it falls through, non-existent
                    ; device selected

               CALL    CMDERR  ; "devices" C thru F

        PWROFF2C:

            ; move result to LATCH1 and write it out

         MOV     P1,ON+1         ; place it on port

         SETB    LATCH1          ; write
        NOP
        NOP
         CLR     LATCH1          ;   it out

    PWROFF3:

     ; return
     RET

FFORWP:

END
```

```
$DEBUG

NAME COMMAND_ERROR

; subroutine CMDERR - used to respond to command error condition
; by echoing back improper command with a question mark
;   POWER module version

; message string:
;     <?><?><CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - May 12, 1986

;
; link references
;

      PUBLIC CMDERR

      EXTRN CODE (OUT_CHAR)

; global variables

      EXTRN DATA (CMD)

$INCLUDE(CONSNT.INC)


; local definitions and variables
;
; const
; var

; begin subroutine

CMDERR_CODE   SEGMENT CODE
RSEG    CMDERR_CODE


    CMDERR:

            ; echo the erroroneous command

          ; MOV    A,CMD            ; first command byte echoed
          ; CALL   OUT_CHAR

          ; MOV    A,CMD+1          ; second command byte echoed
          ; CALL   OUT_CHAR

          MOV    A,#QUESTION      ; question mark
          CALL   OUT_CHAR
          CALL   OUT_CHAR

          MOV    A,#CR
          CALL   OUT_CHAR         ;<CR>


          RET

    RREDMC:

END
```

```
$DEBUG

NAME FORMAT_ERROR

; subroutine FORERR -used to respond to message format
; error condition by sending back error message with a question
; mark
;   STRING optical module / POWER module version
;
; message string:
;
;    <$>FORERR?<CR>

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 13, 1985

;
; link references
;

      PUBLIC FORERR

      EXTRN CODE (PUT_STRING, PUT_CHAR)

; global variables

$INCLUDE(CONSNT.INC)


; local definitions and variables
;
; const

FORERR_MSG    SEGMENT CODE
RSEG   FORERR_MSG

        FORMSG_1:       DB      '$FORMAT?',00H  ; format error message

; var
; begin subroutine

FORERR_CODE   SEGMENT CODE
RSEG    FORERR_CODE


    FORERR:

                ; send the format error message

          MOV    DPTR,#FORMSG_1
          CALL   PUT_STRING

          MOV    A,#CR
          CALL   PUT_CHAR

          RET

    RREROF:

END
```

APPENDIX F

Source Code Listing of the Underwater Hawai'i Programming Language (UHPS)

The source code for program UHPS consists of 15 modules which are listed below in approximate functional category. The include modules (INC) are definitions of constants and macros giving structure to the language. The primitive subroutines (P51) are subroutines which work very close to the hardware registers and do not use the stack for information passing. The subroutines (A51) and subroutine collections (S51) preform higher level functions relatively independent of the special hardware features of the 8051 family such as information type identification and conversion. These are all listed according to the order shown in Table F.1 below:

Table F.1 Listing Order of UHPS Routines

Class	Name	Description
Include files (INC):		
	UHPS	Master include file
	CONSNT	Constant definitions
	DASTRC	Data structure definitions (stack and table)
	RUNMAC	Runtime macro definitions
	LCSTRC	Logic control structure definitions
Primative subroutines (P51):		
	COMLIB	Communications library
	TIMER	Software timing library
	TABLE	Table handling library
	RUNTIM	Runtime library

Subroutines (A51):

INIT3	System initialization
ISALNO	Is alphnumeric?
ISHEX	Is hex?
ASCBIN	ASCII to binary
GFETA	Embedded message

Subroutines (S51):

HEXSUB	Hex to ASCII and ASCII to hex

The listing of the individual modules now follows:

```
; UHPS.INC - include file for forcing inclusion of the standard UHPS
; include files

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


$INCLUDE (CONSNT.INC)
$INCLUDE (RUNMAC.INC)
$INCLUDE (LCSTRC.INC)
```

```
; include CONSNT.INC
;
; file of global constants to be included at assembly time

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 14, 1985

; global definitions
;
; const

; logical state constants

TRUE       EQU   01H    ; logical state
FALSE      EQU   00H    ; logical state

; ASCII constants

CR         EQU   0DH    ; carriage return
LF         EQU   0AH    ; line feed
ESC        EQU   1BH    ; escape
GREATER    EQU   3EH    ; > used as a terminal prompt
DOLLAR     EQU   24H    ; $ used for comm's synch
QUESTION   EQU   3FH    ; ?

BIGA       EQU   41H    ; capital letter A
BIGB       EQU   42H    ; B
BIGC       EQU   43H    ; C
BIGD       EQU   44H    ; D
BIGE       EQU   45H    ; E
BIGF       EQU   46H    ; F
BIGG       EQU   47H    ; G
BIGH       EQU   48H    ; H
BIGI       EQU   49H    ; I
BIGJ       EQU   4AH    ; J
BIGK       EQU   4BH    ; K
BIGL       EQU   4CH    ; L
BIGM       EQU   4DH    ; M
BIGN       EQU   4EH    ; N
BIGO       EQU   4FH    ; O
BIGP       EQU   50H    ; P
```

```
BIGQ      EQU   51H   ; Q
BIGR      EQU   52H   ; R
BIGS      EQU   53H   ; S
BIGT      EQU   54H   ; T
BIGU      EQU   55H   ; U
BIGV      EQU   56H   ; V
BIGW      EQU   57H   ; W
BIGX      EQU   58H   ; X
BIGY      EQU   59H   ; Y
BIGZ      EQU   5AH   ; Z

NULL      EQU   00H   ; ASCII null
NUM0      EQU   30H   ; ASCII number zero
NUM1      EQU   31H   ; one
NUM2      EQU   32H   ; two
NUM3      EQU   33H   ; three
NUM4      EQU   34H   ; four
NUM5      EQU   35H   ; five
NUM6      EQU   36H   ; six
NUM7      EQU   37H   ; seven
NUM8      EQU   38H   ; eight
NUM9      EQU   39H   ; nine
SPACE     EQU   20H   ; ASCII space

; 8051 timer/counter program model and constants

; TCON - timer/counter control/status register definitions

; OVERFLOW FLAG -
;       set on overflow
;       clear by software

;     TF0    EQU    TCON.5        ; timer0
;     TF1    EQU    TCON.7        ; timer1

; RUN CONTROL -
;       set to start
;       clear to stop

;     TR0    EQU    TCON.4
;     TR1    EQU    TCON.6
; INTERRUPT EDGE FLAG (i.e. external interrupt detected)
;       set on interrupt
;        clear by software interrupt handler

;    IE0    EQU    TCON.1
;    IE1    EQU    TCON.3

; INTERRUPT TYPE CONTROL FLAG (i.e. sets interrupt trigger on
;       falling edge or low level)
;       set for falling edge
;       clear for low level

;    IT0    EQU    TCON.0
;    IT1    EQU    TCON.2

; TIMER/COUNTER0

        ; mode control selection

        MOD00   EQU    00000000B     ; 8 bit cntr, /32 prescaler
        MOD10   EQU    00000001B     ; 16 bit cntr
        MOD20   EQU    00000010B     ; 8 bit cntr, autoreload
        MOD30   EQU    00000011B     ; two 8 bit cntr's

        ; counter or timer

      COUNT0      EQU    00000100B     ; select counter
       TIME0      EQU    00000000B     ; select timer

        ; external gate control

      GTON0       EQU    00001000B     ; external gate control ON
      GTOFF0      EQU    00000000B     ; external gate control OFF
```

```
; TIMER/COUNTER1

        ; mode control selection

        MOD01   EQU    00000000B       ; 8 bit cntr, /32 prescaler
        MOD11   EQU    00010000B       ; 16 bit cntr
        MOD21   EQU    00100000B       ; 8 bit cntr, autoreload
        MOD31   EQU    00110000B       ; disrupt counting
                                         ;  [alt   start/stop]

        ; counter or timer

       COUNT1      EQU    01000000B       ; select counter
        TIME1      EQU    00000000B       ; select timer

        ; external gate control

       GTON1       EQU    10000000B       ; external gate control ON
       GTOFF1      EQU    00000000B       ; external gate control OFF

        ; NOTE: timer initiation at assembly time

      ;  EX: MOV    TMOD,MOD10+COUNT0+GTOFF0
                                 ; 8-bit COUNTER0 w/o external gate
; NAME DATA_STRUCTURES
;
; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


; link references
;
    ; NONE




; note: this is a collection of macros that impliment useful data
; structures such as stacks (FILO) and tables (random access).

; begin data structure macros

        ; MUPDATE - macro to write to a table

    %*DEFINE(MUPDATE(TABLE,OFFSET,VALUE)) LOCAL LABEL
     (
          MOV    A,#%TABLE          %'point to table entry'
          ADD    A,%OFFSET          %'add offset'
          MOV    R0,A               %'point to position in table'
                                    %'load pointer'
          MOV    @R0,%VALUE         %'fetch value'
                                    %'write value into table'
     )

        ; MFETCH - macro to read from a table

    %*DEFINE(MFETCH(TABLE,OFFSET,VALUE)) LOCAL LABEL
     (
          MOV    A,#%TABLE          ;'point to beginning of table'
          ADD    A,%OFFSET          ;'location = table +offset'
          MOV    R0,A               ;'ptr --> location'
          MOV    %VALUE,@R0         ;'retrieve value from table'
     )
    ; stack macros - current implimentation treats stacks as FILO
    ; tables with a global pointer around always pointing to the
    ; next empty location.  this pointer is one byte long and thus
    ; limits the stack length to 128 bytes long.  a future upgrade
    ; would be to make the pointer two bytes long thus allowing it
    ; to point to any part of data memory.  additionally, the stack
    ; is treated as a push down stack rather than a push up stack
    ; like the system stack and thus it grows downward in memory
    ; rather than upward from its root location.

    ; MPSH - macro to push byte (value) onto stack

  %*DEFINE(MPSH(USTACK,STKPTR,VALUE)) LOCAL LABEL
  (
     %MUPDATE(%USTACK,%STKPTR,%VALUE)
      DEC    %STKPTR
  )

    ; MPUL - macro to pull byte (value) off of stack

  %*DEFINE(MPUL(USTACK,STKPTR,VALUE)) LOCAL LABEL
  (
      INC    %STKPTR
     %MFETCH(%USTACK,%STKPTR,%VALUE)
  )

; end data structure macros
; NAME RUN_TIME_MACROS

; note: this is a collection of macros that mirror the decision logic
; primatives of the run time library RUNLIB.P51. their function is to
; simplify the calling of the run time primatives through the use of
; additional structured logic flow macros found in LCSTRC.INC.
;
; additionally, XDATA oriented versions of the above
; mentioned macros are included as well as the seven macros
; EPOP, EPUSH, EMOV, TMOV, FMOV, TAMOV, and FAMOV to
; facilitate the semiautomatic conversion of routines
; originally written using DATA instead of XDATA memory.

; author - John F. Babson, University of Hawaii Physics

; revision date - Sept. 16, 1985

;
; link references
;

    ; NONE




; begin decision macros

        ; in the following, X and Y are two byte values to be compared.
        ; the form MXOPY is used when X and Y are either DATA bytes or
        ; constants. the form EXOPY is used when both X and Y
        ; are XDATA bytes.
        ; the forms EXOPK or EKOPY are used when either X or Y
        ; is either a constant ("K" for constant) or DATA byte and the
        ; other one is an XDATA byte.
```

```
                                              ;     X     >     Y     ?
**********************************************************************

    %*DEFINE(MXGTY(X,Y)) LOCAL LABEL
     (
          MOV     R0,%X
          MOV     R1,%Y
          CALL    XGTY
     )

    %*DEFINE(EXGTY(X,Y)) LOCAL LABEL
     (
         %TMOV(R0,%X)
         %TMOV(R1,%Y)
          CALL    XGTY
     )

    %*DEFINE(EXGTK(X,K)) LOCAL LABEL
     (
         %TMOV(R0,%X)
          MOV     R1,%K
          CALL    XGTY
     )

    %*DEFINE(EKGTY(K,Y)) LOCAL LABEL
     (
          MOV     R0,%K
         %TMOV(R1,%Y)
          CALL    XGTY
     )
```

```
                                        ;     X     <     Y     ?
******************************************************************

   %*DEFINE(MXLTY(X,Y)) LOCAL LABEL
    (
        MOV     R0,%X
        MOV     R1,%Y
        CALL    XLTY
    )

   %*DEFINE(EXLTY(X,Y)) LOCAL LABEL
    (
        %TMOV(R0,%X)
        %TMOV(R1,%Y)
        CALL    XLTY
    )

   %*DEFINE(EXLTK(X,K)) LOCAL LABEL
    (
        %TMOV(R0,%X)
        MOV     R1,%K
        CALL    XLTY
    )

   %*DEFINE(EKLTY(K,Y)) LOCAL LABEL
    (
        MOV     R0,%K
        %TMOV(R1,%Y)
        CALL    XLTY
    )
```

```
                                        ;      X        =       Y       ?
*****************************************************************************

    %*DEFINE(MXEQY(X,Y)) LOCAL LABEL
     (
          MOV     R0,%X
          MOV     R1,%Y
          CALL    XEQY
     )

    %*DEFINE(EXEQY(X,Y)) LOCAL LABEL
     (
          %TMOV(R0,%X)
          %TMOV(R1,%Y)
          CALL    XEQY
     )

    %*DEFINE(EXEQK(X,K)) LOCAL LABEL
     (
          %TMOV(R0,%X)
          MOV     R1,%K
          CALL    XEQY
     )

    %*DEFINE(EKEQY(K,Y)) LOCAL LABEL
     (
          MOV     R0,%K
          %TMOV(R1,%Y)
          CALL    XEQY
     )
```

```
                                                    ;      X      !=      Y      ?
****************************************************************************************

%*DEFINE(MXNEY(X,Y)) LOCAL LABEL
 (
     MOV     R0,%X
     MOV     R1,%Y
     CALL    XNEY
 )

%*DEFINE(EXNEY(X,Y)) LOCAL LABEL
 (
     %TMOV(R0,%X)
     %TMOV(R1,%Y)
     CALL    XNEY
 )

%*DEFINE(EXNEK(X,K)) LOCAL LABEL
 (
     %TMOV(R0,%X)
     MOV     R1,%K
     CALL    XNEY
 )

%*DEFINE(EKNEY(K,Y)) LOCAL LABEL
 (
     MOV     R0,%K
     %TMOV(R1,%Y)
     CALL    XNEY
 )
```

```
                                        ;       X       =>       Y       ?
*****************************************************************************

   %*DEFINE(MXGEY(X,Y)) LOCAL LABEL
    (
         MOV     R0,%X
         MOV     R1,%Y
         CALL    XGEY
    )

   %*DEFINE(EXGEY(X,Y)) LOCAL LABEL
    (
         %TMOV(R0,%X)
         %TMOV(R1,%Y)
         CALL    XGEY
    )

   %*DEFINE(EXGEK(X,K)) LOCAL LABEL
    (
         %TMOV(R0,%X)
         MOV     R1,%K
         CALL    XGEY
    )

   %*DEFINE(EKGEX(K,Y)) LOCAL LABEL
    (
         MOV     R0,%K
         %TMOV(R1,%Y)
         CALL    XGEY
    )
```

```
                                             ;     X     <=     Y     ?
*****************************************************************

    %*DEFINE(MXLEY(X,Y)) LOCAL LABEL
     (
         MOV    R0,%X
         MOV    R1,%Y
         CALL   XLEY
     )

    %*DEFINE(EXLEY(X,Y)) LOCAL LABEL
     (
         %TMOV(R0,%X)
         %TMOV(R1,%Y)
         CALL   XLEY
     )

    %*DEFINE(EXLEK(X,K)) LOCAL LABEL
     (
         %TMOV(R0,%X)
         MOV    R1,%K
         CALL   XLEY
     )

    %*DEFINE(EKLEY(K,Y)) LOCAL LABEL
     (
         MOV    R0,%K
         %TMOV(R1,%Y)
         CALL   XLEY
     )

; end --- the decision macros
; begin XDATA PUSH, POP and MOV macros

      ; move an XDATA source byte (X) to a DATA destination (D)

    %*DEFINE(TMOV(D,X)) LOCAL LABEL
     (
          MOV     DPTR,#%X
          MOVX    A,@DPTR
          MOV     %D,A
     )

      ; move a DATA source byte (D) to an XDATA destination (X)

    %*DEFINE(FMOV(X,D)) LOCAL LABEL
     (
          MOV     A,%D
          MOV     DPTR,#%X
          MOVX    @DPTR,A
     )

      ; move an XDATA source byte (X) to the A register

    %*DEFINE(TAMOV(X)) LOCAL LABEL
     (
          MOV     DPTR,#%X
          MOVX    A,@DPTR
     )

      ; move the A register to an XDATA destination byte (X)

    %*DEFINE(FAMOV(X)) LOCAL LABEL
     (
          MOV     DPTR,#%X
          MOVX    @DPTR,A
     )
```

```
  ; push an XDATA byte (X) onto the system stack

%*DEFINE(EPUSH(X)) LOCAL LABEL
 (
     MOV     DPTR,#%X
     MOVX    A,@DPTR
     MOV     STACC,A
     PUSH    STACC
 )

  ; pop an XDATA byte (X) off the system stack

%*DEFINE(EPOP(X)) LOCAL LABEL
 (
     POP     STACC
     MOV     A,STACC
     MOV     DPTR,#%X
     MOVX    @DPTR,A
 )

  ; mov an XDATA byte (X) to an XDATA byte(y)

%*DEFINE(EMOV(X,Y)) LOCAL LABEL
 (
     MOV     DPTR,#%Y
     MOVX    A,@DPTR
     MOV     DPTR,#%X
     MOVX    @DPTR,A
 )

; end --- the XDATA macros
; begin  increment/decrement  XDATA  macros

       ; increment an XDATA byte (X)

     %*DEFINE(EINC(X)) LOCAL LABEL
      (
          MOV     DPTR,#%X
          MOVX    A,@DPTR
           INC    A
          MOVX    @DPTR,A
      )

       ; decrement an XDATA byte (X)

     %*DEFINE(EDEC(X)) LOCAL LABEL
      (
          MOV     DPTR,#%X
          MOVX    A,@DPTR
           DEC    A
          MOVX    @DPTR,A
      )

; end --- the increment/decrement XDATA macros
; begin arithmetic XDATA macros

     ; straight addition of XDATA byte to <A>

   %*DEFINE(EADD(X)) LOCAL LABEL
    (
         MOV    R0,A
         MOV    DPTR,#%X
         MOVX   A,@DPTR
         MOV    R1,A
         MOV    A,R0
         ADD    A,R1
    )

     ; addition with carry of XDATA byte to <A>

   %*DEFINE(EADDC(X)) LOCAL LABEL
    (
         MOV    R0,A
         MOV    DPTR,#%X
         MOVX   A,@DPTR
         MOV    R1,A
         MOV    A,R0
         ADDC   A,R1
    )

     ; subtract with borrow of XDATA byte to <A>

   %*DEFINE(ESUBB(X)) LOCAL LABEL
    (
         MOV    R0,A
         MOV    DPTR,#%X
         MOVX   A,@DPTR
         MOV    R1,A
         MOV    A,R0
         SUBB   A,R1
    )

; end --- the arithmetic XDATA macros
; begin the logic XDATA macros

        ; logical AND of XDATA byte with <A>

      %*DEFINE(EAND(X)) LOCAL LABEL
       (
            MOV    R0,A
            MOV    DPTR,#%X
            MOVX   A,@DPTR
            MOV    R1,A
            MOV    A,R0
            EAND   A,R1
       )

        ; logical OR of XDATA byte with <A>

      %*DEFINE(EOR(X)) LOCAL LABEL
       (
            MOV    R0,A
            MOV    DPTR,#%X
            MOVX   A,@DPTR
            MOV    R1,A
            MOV    A,R0
            EOR    A,R1
       )

; end --- the logical XDATA macros
; NAME LOGIC_CONTROL_STRUCTURES
;
; link references
;
              ; NONE


; note: this is a collection of macros that impliment the
; standard logic control structures as found in such languages as
; C, RATFOR, and PASCAL.

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986



; begin control macros


    ; if then or else

      ; if (CONDITION) {      }

     %*DEFINE(MIF(X,CONDITION,Y,ELSELAB)) LOCAL LABEL
      (
         %%CONDITION(%X,%Y)
          JNC    %ELSELAB
      )

      ; else {        }

     %*DEFINE(MELSE(ELSELAB,ENDLAB)) LOCAL LABEL
      (
                 JMP    %ENDLAB
          %ELSELAB:
      )
```

```
; while loop

  ; while (CONDITION) {

 %*DEFINE(MWHILE(X,CONDITION,Y,BEGIN,END)) LOCAL LABEL
  (
      %BEGIN:
           %%CONDITION(%X,%Y)
            JNC     %END
  )

    ; } */ end the while /*

 %*DEFINE(MWEND(BEGIN,END)) LOCAL LABEL
  (
            JMP     %BEGIN
      %END:
  )


; repeat until

  ; repeat {

 %*DEFINE(MREPEAT(BEGIN)) LOCAL LABEL
  (
      %BEGIN:
  )

  ; until (CONDITION)

 %*DEFINE(MUNTIL(X,CONDITION,Y,BEGIN,END)) LOCAL LABEL
  (
           %%CONDITION(%X,%Y)
            JNC     %BEGIN
      %END:
  )


; end --- the control macros
```

```
NAME COMMUNICATIONS_LIBRARY

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 31, 1985

; link references

    PUBLIC PUT_CRLF, PUT_STRING, PUT_CHAR, GET_CHAR
    PUBLIC OUT_CRLF, OUT_STRING, OUT_CHAR, IN_CHAR
    PUBLIC PUT_DATA_STR, GET_NUM, ECHO

    EXTRN CODE (MDELAY)

; This is the library of communications primatives used to interface
; with an external communication system for receiving
; commands and responding with replies.  The principle routines
; are GET_CHAR and PUT_CHAR which fetch and output a single
; ASCII character at a time respectively.  These are
; primative routines and thus, in this context do not
; pass information to and from the stack upon being called but
; rather do so using the following convention:
;
;       (A)         contains input/output character
;       (R0)        contains data string address
;       (R1)        contains length of data string
;       DPTR        address pointer to code string

; local definitions and variables
;
; const

     CR     EQU    0DH    ; carriage return
     LF     EQU    0AH    ; line feed

; var
; begin library

COMLIB_CODE   SEGMENT CODE
RSEG   COMLIB_CODE

USING  0


        ; primative PUT_CRLF
        ;
         ; This routine outputs a Carriage Return and
        ; a Line Feed

       PUT_CRLF:


            MOV     A,#CR
            CALL    PUT_CHAR
            MOV     A,#LF
            CALL    PUT_CHAR

            RET

        ; primative OUT_CRLF
        ;
        ; OUT_CRLF version of PUT_CRLF

       OUT_CRLF:


            MOV     A,#CR
            CALL    OUT_CHAR
            MOV     A,#LF
            CALL    OUT_CHAR

            RET
```

```
; primative PUT_STRING
;
; Routine outputs a null-terminated string located
; in CODE memory, whose address is given in DPTR.

PUT_STRING:

        CLR     A
        MOVC    A,@A+DPTR
        JZ      EXIT
        CALL    PUT_CHAR
        INC     DPTR
        JMP     PUT_STRING

EXIT:

        RET

; primative OUT_STRING
;
; OUT_CHAR based version of PUT_STRING

OUT_STRING:

        CLR     A
        MOVC    A,@A+DPTR
        JZ      FINISH
        CALL    OUT_CHAR
        INC     DPTR
        JMP     OUT_STRING

FINISH:

        RET
```

```
; primative PUT_DATA_STR
;
; Routine outputs a string located in DATA memory,
; whose address is in R1 and its length in R2.

PUT_DATA_STR:

        MOV     A,@R1
        CALL    PUT_CHAR
        INC     R1
        DJNZ    R2,PUT_DATA_STR

        RET




; primative PUT_CHAR
;
; This routine outputs a single character to console.
; The character is given in A.

PUT_CHAR:

        JNB     TI,$
        CLR     TI
        MOV     SBUF,A

        RET
```

```
; primative OUT_CHAR
;
; this routine is a variation on PUT_CHAR which toggles
; a strobe line defined by the clearing of both the RD
; and WR lines for a wait period of at least 3 microseconds
; prior to outputting a character on the serial I/O line.
; this protocal is necessary in order to control the
; Computrol model CM-500K 500 kilobaud coaxial half duplex
; modem.
;

OUT_CHAR:

        CLR     RD              ; set trans
        CLR     WR              ;   strobe low

        NOP                     ; wait at
        NOP                     ;   least 3
        NOP                     ;     microseconds

        JNB     TI,$            ; output
        CLR     TI              ;   the
        MOV     SBUF,A          ;     character

        SETB    WR              ; set trans
        SETB    RD              ;   strobe high

        RET
```

```
 ; primative GET_NUM
 ;
   ; This routine gets a 4 character string from console
  ; and stores it in memory at the address given in R0.
  ; If a ^X is received, routine starts over again.

GET_NUM:

      MOV    R2,#4        ; set up string length as 4
      MOV    R1,AR0       ; R0 value may be needed for restart

GET_LOOP:

      CALL   GET_CHAR

  ; Next 4 instructions handle ^X- the routine starts
 ; over if received

      CLR    ACC.7              ; clear the parity bit
      CJNE   A,#18H,GO_ON       ; if not ^X- go on
      CALL   PUT_CRLF
      JMP    GET_NUM

GO_ON:

      MOV    @R1,A
      INC    R1
      DJNZ   R2,get_loop

      RET
 ; primative GET_CHAR
;
  ; This routine gets a single character from console.
 ; The character is returned in A.

GET_CHAR:

       JNB    RI,$
       CLR    RI
       MOV    A,SBUF
       ANL    A,#01111111B   ; strip off any parity bit to ASCII
                             ;   input character

       RET
```

```
; primative IN_CHAR
;
; this routine is a variation of GET_CHAR which has a built
; in 1 millisecond delay between each check for a new input
; character.  two counters (R6 and R7) are loaded in advance
; of calling an input subroutine providing a time out facility
; that will force a return should the input not be completed
; within a given period of time.
;
; calling sequence
;
;     LOCOUNT EQU    R6
;     HICOUNT EQU    R7
;     ..................
;       MOV     LOCOUNT,#200    ; 200 ms
;       MOV     HICOUNT,#25     ;  X 25 = 5 sec
;
; finally call input routine which uses IN_CHAR

IN_CHAR:

        MOV     B,#00H      ; timeout status flag starts as FALSE

IN_CHAR1:

      JNB    RI,WAIT
      CLR    RI
      MOV    A,SBUF
      RET

         WAIT:    ; wait 1ms before looking for another character -
       ; decrement timeout counter and return if it underflows

         CALL    MDELAY          ; 1ms delay
           DJNZ    R6,IN_CHAR1  ; not ready, loop 4 char again
            MOV     R6,#200       ; reload lower byte of counter
           DJNZ    R7,IN_CHAR1  ; not zero, loop 4 char again
           MOV     B,#01H         ; timeout status TRUE on return
            RET     ; when both counter byts clear, timeout
```

```
     ; primative ECHO
    ;
       ; Routine echoes all character input from the console
     ; one at a time.
    ;

   ECHO:

         CALL   GET_CHAR
         CALL   PUT_CHAR

         RET

; end library


END
```

```
NAME SOFTWARE_TIMER

; primatives for software timing purposes:
;
; DELAY   - 4 microsec delay
; MDELAY  - 1 millisec delay
; SDELAY  - 1 second delay
; MINDEL  - 1 minute delay
; HDELAY  - 1 hour delay

; author - John F. Babson, University of Hawaii Physics
; revision date - Oct. 21, 1985

; calling sequence:
;
; EX:   CALL   SDELAY   ; 1 second delay
;
;  EX: to produce a 5 millisecond delay
;
;      MLCNT     EQU     R4      ; millisec counter
;
;      MOV       MLCNT,#5H
;
;      FIVECNT:
;
;            CALL   MDELAY
;            DJNZ   MLCNT,FIVECNT
;
;             RET

; programming note:  the registers here could be used in a
; more efficient manner with registers needed only to hold
; count information as to the number of 4micro (R2),
; milli (R3), second (R4), minute (R5), or hour (R6) counts to delay

; link references
;

     PUBLIC DELAY, MDELAY, SDELAY, MINDEL, HDELAY
```

```
; programming model for all of the primative routines

;     A      character  i/o  buffer
;      R0      data string address
;      R1      data string length
;     DPTR  code  string  address
;      R2      microsecond clock
;      R3      microsecond counter
;      R4      millisecond clock
;      R5      millisecond counter
;     R6     second clock
;     R7     second counter

; local  definitions  and  variables
;
; const


; var

        MRCLK        EQU     R2       ; microsecond clock
        MRCNT        EQU     R3       ; microsecond counter
                                        ; counts  no.  of  250  microsec's
        MLCLK        EQU     R4       ; millisecond clock
        MLCNT        EQU     R5       ; millisecond counter
                                        ; counts  no.  of  1000  millisec's
        SCCLK        EQU     R6      ; second clock
        SCCNT        EQU     R7      ; second counter
```

```
; begin library

STIMER_CODE   SEGMENT CODE
RSEG   STIMER_CODE

     DELAY:          ; 6 microsecond delay

             ; 2 microseconds for CALL (either ACALL or LCALL) plus
            ; 2 microseconds for RET

          NOP     ; plus 2 microseconds for NOP's
        NOP

        RET

     MDELAY:          ; 1 millisecond delay

        MOV    MRCNT,#2          ; (2 + 496 + 3 MOV) x 2
                                 ;     = 1002 microsec = 1 millisec

    MDELAY1:

        MOV    MRCLK,#62         ; 62 x 8 = 496 microsec

    MDELAY2:

        CALL   DELAY             ; 6 microsec
       DJNZ   MRCLK,MDELAY2 ; 2 microsec => 8 microsec

       DJNZ   MRCNT,MDELAY1 ; 2 microsec => 2 microsec

        RET

     SDELAY:          ; 1 second delay

        MOV    MLCNT,#4          ; 4 X 250 millisec = 1 second

    SDELAY1:

        MOV    MLCLK,#250        ; 250 millisec
```

```
SDELAY2:

        CALL   MDELAY
        DJNZ   MLCLK,SDELAY2
        DJNZ   MLCNT,SDELAY1

        RET

 MINDEL:         ; 1 minute delay

          MOV    SCCLK,#60        ; 60 x 1 sec = 1 min

MINDEL1:

        CALL   SDELAY
        DJNZ   SCCLK,MINDEL1

        RET

 HDELAY:         ; 1 hour delay

          MOV    SCCNT,#60        ; 60 x 1 min = 1 hour

HDELAY1:

        CALL   MINDEL
        DJNZ   SCCNT,HDELAY1

        RET


; end library


END
```

```
NAME TABLE_LIBRARY
;
; collection of primatives for handling XDATA tables
;

; author - John F. Babson, University of Hawaii Physics

; revision date - Sept. 18, 1985

;
; link references
;

     PUBLIC POINT, STORE, FETCH

; programming model
;
;       R0      lower address byte
;       R1      upper address byte
;       R2      table offset

     LOWER      EQU   R0
     UPPER      EQU   R1
     OFFSET     EQU   R2

; calling sequence (note the * indicates a macro since MPL
; will expand any use of the real macro symbol here even
; though this line is a comment)
;
;    EX:   *TMOV (R2,OFFSET)
;          MOV    DPTR,#TABLE
;          CALL   POINT
;
;          MOV    DPTR,#CELL        CALL    FETCH
;          MOVX   A,@DPTR     or    MOV     DPTR,#CELL
;          CALL   STORE             MOVX    @DPTR,A
;
```

```
; begin library

TABLE_CODE SEGMENT CODE
RSEG TABLE_CODE

        ; point to first entry in XDATA table no longer than 256
        ; bytes long and ADD known offset to the pointer

       POINT:

            CLR    C             ; clear carry setting known state
            MOV    A,DPL         ; add offset
            ADDC   A,OFFSET  ;   to lower
            MOV    DPL,LOWER ;   byte of DPTR

            MOV    A,DPH         ; increment upper
            ADDC   A,#00H    ;   byte of DPTR
            MOV    DPH,UPPER ;     if any carry
            RET

         ; store byte currently in <A> at the location pointed to by POINT

       STORE:

            MOV    DPL,LOWER        ; retrieve address
            MOV    DPH,UPPER        ;   within TABLE
            MOVX   @DPTR,A          ; store value there
            RET

         ; fetch byte at location pointed to by POINT and place it in <A>

       FETCH:

            MOV    DPL,LOWER        ; retrieve address
            MOV    DPH,UPPER        ;   within TABLE
            MOVX   A,@DPTR          ; fetch value there
            RET

; end --- of the library

END
```

```
NAME RUN_TIME_LIBRARY
;
; link  references
;
    PUBLIC XGTY, XLTY, XEQY, XNEY, XGEY, XLEY

; note: this is a collection of logical condition primatives used for
; making logical comparisions such as is X > Y ? etc.  the calling
; sequence in all cases is as follows:
;
;     MOV    R0,X
;     MOV    R1,Y
;     CALL   XGTY
;
; i.e. the first quantity X is loaded into the R0 register,
; the second quantity Y is loaded into the R1 register.
; finally, the result of the comparision, either TRUE = 1
; or FALSE = 0 is returned in the carry flag (C).

; local  definitions  and  variables
;
; const



; var

      LOBYTE  EQU    R0      ; lower byte for DPTR incrementation
     UPBYTE  EQU    R1     ; upper "   "   "   "

    RUNLIB_DATA    SEGMENT DATA
    RSEG   RUNLIB_DATA

          X:     DS     1     ; X and Y are
           Y:     DS     1     ; comparision values
```

```
; begin library

RUNLIB_CODE             SEGMENT CODE
RSEG RUNLIB_CODE


        ; X > Y ?  this is the most basic of the comparision primatives.
        ; it is used by all of the other comparision primatives.

    XGTY:

            MOV    A,R1          ; Y
            CPL    A
            MOV    R1,A          ; !Y
            MOV    A,R0          ; X
            CLR    C
            ADDC   A,R1
            RET

      XLTY:   ; untried, this is independent of the others

            MOV    A,R0          ; X
            CPL    A
            MOV    R0,A          ; !X
            MOV    A,R1          ; Y
            CLR    C
            ADDC   A,R0
            RET
```

```
; X = Y ?

XEQY:

        MOV    X,R0           ; save X
        MOV    Y,R1           ; save Y
        CALL   XGTY
        JC     EQNO
        MOV    R0,Y
        MOV    R1,X
        CALL   XGTY
        JC     EQNO
        SETB   C             ; C is set thus TRUE
        RET

EQNO:
        CLR    C              ; C is not set thus FALSE
        RET


; X != Y ?

XNEY:

        MOV    X,R0           ; save X
        MOV    Y,R1           ; save Y
        CALL   XGTY
        JC     EQYES
        MOV    R0,Y
        MOV    R1,X
        CALL   XGTY
        JC     EQYES
        RET                  ; C is cleared thus FALSE

EQYES:

        RET                  ; C is set thus TRUE
```

```
; X => Y ?

XGEY:

        MOV     X,R0            ; save X
        MOV     Y,R1            ; save Y
        CALL    XGTY
        JC      GEYES
        MOV     R0,X
        MOV     R1,Y
        CALL    XEQY
        JC      GEYES
        RET                     ; C is cleared thus FALSE

GEYES:

        RET                     ; C is set thus TRUE

; X <= Y ?

XLEY:

        MOV     X,R0            ; save X
        MOV     Y,R1            ; save Y
        CALL    XGTY
        JNC     LEYES
        MOV     R0,X
        MOV     R1,Y
        CALL    XEQY
        JC      LEYES1
        RET

LEYES:
        SETB    C               ; LE True

LEYES1:
        RET
```

```
 ; second, here is a primative to decrement the data pointer
 ; (DPTR) since none exists in symmetry to the INC DPTR
 ; instruction

DDPTR:

     MOV    UPBYTE,DPH      ; fetch DPTR
     MOV    LOBYTE,DPL      ;    in two bytes
     DJNZ   LOBYTE,NOBORROW ; decrement lower byte
     DEC    UPBYTE          ; borrow 1 from upbyte

  NOBORROW:

     MOV    DPL,LOBYTE
     MOV    DPH,UPBYTE


; end --- of the library

END
```

```
NAME INITIAL

; intrinsic routine for initializing the 8051 family microcontroller
; both the system stack and the serial I/O port are initialized

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986

; link references
;

        PUBLIC  INITIAL

; local definitions and variables
;
; const

       AUTORLD    EQU 00100000B  ; auto-reload mode for timer
        BAU300    EQU 99H        ; 300 baud timer
       SERPORT    EQU 11011010B  ; Serial Port

; var

DSEG at    8

STACK:     DS   24    ; at power-up, the stack pointer
                      ; is initialized to point here.
                      ; 24 byte space reserved.
; begin subroutine

INITIAL_PROCESSOR segment CODE
RSEG INITIAL_PROCESSOR

        INITIAL:

                ; This is the initializing section. Execution always
               ; starts at address 0 on power-up.

            MOV    TMOD,#AUTORLD  ; set timer mode to auto-reload
            MOV    TH1,#BAU300    ; set timer for 300 BAUD
                                  ; (self determined)
            MOV    SCON,#SERPORT  ; prepare the Serial Port
            SETB   TR1            ; start clock

            RET

        LAITINI:

END
```

```
$DEBUG

NAME IS_IT_ALPHA_NUMERIC

; SUBROUTINE ISALNO
;
; routine to check whether or not a candidate character
; is a valid ASCII alphanumeric value or not
;
; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


; link references
;

    PUBLIC  ISALNO
   EXTRN CODE (XGEY, XLEY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const

; var

ISALNO_DATA SEGMENT DATA
RSEG ISALNO_DATA

    CISANO1:   DS    1     ; return
    CISANO2:   DS    1     ;  address
    CHAR:      DS    1     ; character buffer
    EFLAG:     DS    1     ; condition flag
```

```
; begin  subroutine

ISALNO_CODE SEGMENT CODE
RSEG ISALNO_CODE

     ISALNO:

            POP     CISANO1 ; save return
           POP     CISANO2 ;   address

             POP      CHAR     ; the candidate alphanumeric character

            ; set error flag EFLAG = #FALSE
              ; this is the default condition for
             ; the following comparison

          MOV     EFLAG,#FALSE

            ; now, successively compare CHAR to
              ; see if it is a valid ASCII alphanumeric character
            ; setting EFLAG = #TRUE iff it is

            ; case - is it a number, then set eflag TRUE

         %MIF (CHAR,MXGEY,#NUM0,ISALNO1)

             %MIF (CHAR,MXLEY,#NUM9,COMPEND)

                    MOV     EFLAG,#TRUE

          ISALNO1:

             ; case - is it a capital letter, then set eflag TRUE

         %MIF (CHAR,MXGEY,#BIGA,ISALNO2)

             %MIF (CHAR,MXLEY,#BIGZ,COMPEND)

                    MOV     EFLAG,#TRUE
     ISALNO2:

         ; case - is it a small letter, then set eflag TRUE

    %MIF(CHAR,MXGEY,#LETA,ISALNO3)

         %MIF (CHAR,MXLEY,#LETZ,COMPEND)

                 MOV     EFLAG,#TRUE

     COMPEND:        ; end comparisons

       ; now, return

     PUSH    EFLAG      ; return  condition

     PUSH    CISANO2    ; restore  return
     PUSH    CISANO1    ;   address

     RET                ; return

  ONLASI:

END
```

```
$DEBUG

NAME IS_IT_HEX

; SUBROUTINE ISHEX
;
; routine to check whether or not a candidate character
; is a valid ASCII hex value or not
;
; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


; link references
;

      PUBLIC  ISHEX
     EXTRN CODE (XGEY, XLEY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const

; var

ISHEX_DATA SEGMENT DATA
RSEG ISHEX_DATA

     CISHEX1:    DS     1       ; return
     CISHEX2:    DS     1      ;   address
     CHAR:       DS      1       ; character buffer
     EFLAG:      DS      1       ; condition flag
```

```
; begin  subroutine

ISHEX_CODE SEGMENT CODE
RSEG ISHEX_CODE

      ISHEX:

              POP     CISHEX1 ; save return
              POP     CISHEX2 ;   address

              POP     CHAR    ; the candidate hex character

              ; set error flag EFLAG = #FALSE
                ; this is the default condition for
               ; the following comparison

             MOV     EFLAG,#FALSE

              ; now, successively compare CHAR to
               ; see if it is a valid ASCII hex character
              ; setting EFLAG = #TRUE iff it is

          %MIF (CHAR,MXGEY,#NUM0,COMPEND)

              %MIF (CHAR,MXLEY,#BIGF,COMPEND)

                   %MIF (CHAR,MXLEY,#NUM9,ISHEX1)

                       MOV    EFLAG,#TRUE

                  %MELSE (ISHEX1,COMPEND)

                       %MIF(CHAR,MXGEY,#BIGA,COMPEND)

                           MOV    EFLAG,#TRUE

             COMPEND:        ; end comparisons
```

```
        ; now, return

        PUSH    EFLAG   ; return condition

        PUSH    CISHEX2 ; restore return
       PUSH    CISHEX1 ;   address

       RET             ; return

    XEHSI:

END
```

```
NAME ASCII_TO_BINARY

; SUBROUTINE ASCBIN
;
; routine to convert ascii hex numbers to their absolute binary
; values.  ascii input should first be tested for validity using
; ISHEX.
;
; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


; link references
;

        PUBLIC  ASCBIN, STAK
      EXTRN CODE (XLEY)

; global constants and variables

$INCLUDE (UHPS.INC)

; local definitions and variables
;
; const
; var

ASCBIN_DATA SEGMENT DATA
RSEG ASCBIN_DATA

     STAK:        DS      1       ; stack scratch buffer

ASCBIN_XDATA SEGMENT XDATA PAGE
RSEG ASCBIN_XDATA

             ORG    100H        ; beginning of external data memory
                                ; page

     CASCBIN1:  DS      1       ; return
     CASCBIN2:  DS      1       ;   address
      ASCII:    DS      1       ; input ASCII byte
     BINARY:    DS      1       ; output BINARY byte

; begin subroutine

ASCBIN_CODE SEGMENT CODE
RSEG ASCBIN_CODE

     ASCBIN:

          POP    STAK        ; save return
          MOV    A,STAK
          MOV    DPTR,#CASCBIN1
          MOVX   @DPTR,A

          POP    STAK        ;   address
          MOV    A,STAK
          MOV    DPTR,#CASCBIN2
          MOVX   @DPTR,A

          POP    STAK        ; the candidate hex character
          MOV    A,STAK
          MOV    DPTR,#ASCII
          MOVX   @DPTR,A
```

```
   ; check if ASCII byte is a number [0..9]
   ; else it is a letter [A..F]

;MIF (ASCII,MXLEY,#NUM9,ASCBIN1)

   ; hand coded kludge for 'if' to use external data space

MOV    DPTR,#ASCII
MOVX   A,@DPTR
MOV    R0,A

MOV    R1,#NUM9
CALL   XLEY

JNC    ASCBIN1

  ; end the 'if' kludge

        ; convert ascii number [0..9]

     MOV     DPTR,#ASCII
     MOVX    A,@DPTR    ; binary
     SUBB    A,#2FH     ;  = number - 2FH
     MOV     DPTR,#BINARY
     MOVX    @DPTR,A

%MELSE (ASCBIN1,ASCBIN2)

        ; convert ascii letter [A..F]


     MOV     DPTR,#ASCII
     MOVX    A,@DPTR    ; binary
     SUBB    A,#41H     ;  = letter - 41H + 0AH
     ADDC    A,#0AH
     MOV     DPTR,#BINARY
     MOVX    @DPTR,A
```

```
        ASCBIN2:        ; endif

         ; now, return

       MOV    DPTR,#BINARY
       MOVX   A,@DPTR
       MOV    STAK,A
       PUSH   STAK        ; return value

       MOV    DPTR,#CASCBIN2
       MOVX   A,@DPTR
       MOV    STAK,A
       PUSH   STAK        ; restore address

       MOV    DPTR,#CASCBIN1
       MOVX   A,@DPTR
       MOV    STAK,A
       PUSH   STAK        ;  address

       RET                ; return

   NIBCSA:

END
```

```
NAME GFETA

; message to extraterrestrial visitors should instrument survive
; at bottom of the ocean while  us surface creatures and our
; civilization does not

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1985


GFETA_CODE   SEGMENT CODE
RSEG   GFETA_CODE

        DB    '3 31 314 3141 31415 314159  '

       DB    '0 1 2 3 4 5 6 7 8 9  '

      DB   'GREETINGS FROM PLANET EARTH FROM '
      DB    'JOHN BABSON, '
      DB    'DAVE HARRIS, '
      DB    'JOHN LEARNED, '
      DB    'SHIGE MATSUNO, '
      DB    'YOSHIKO MIYAKOSHI, '
      DB    'DAN OCONNOR, '
      DB    'VIC STENGER, '
      DB    'CHUCK WILSON  '

      DB    'A B C D E F G H I J K L M'
      DB    'N O P Q R S T U V W X Y Z  '

        DB    '3 31 314 3141 31415 314159  '

END
```

```
NAME HEXADECIMAL_BINARY_SUBROUTINES

; this file is a collection of the routines for converting from binary
; to hex and back:
;
;      HEXBIN, HEXNIB --> convert hex to binary
;      BINHEX, NIBHEX --> convert binary to hex

; author - John F. Babson, University of Hawaii Physics
; revision date - Aug 1986


; library link references

    EXTRN CODE (XGTY, XNEY, XEQY, XGEY, XLEY)

    PUBLIC HEXBIN, BINHEX


; the following include files are common to all of the subroutines in
; this file and should be moved with any subroutine extracted from
; this file.

$INCLUDE(RUNMAC.INC)
$INCLUDE(LCSTRC.INC)
$INCLUDE(CONSNT.INC)



; subroutine hexbin
;
; link references
;
    ; EXTRN CODE (XGTY, XEQY, XNEY, XGEY, XLEY)


; local definitions and variables
;
; const
; var

HEXBIN_DATA    SEGMENT DATA
RSEG   HEXBIN_DATA

      LOWER:       DS      1      ; lower hex character
      UPPER:       DS      1      ; upper hex character
      BINARY:      DS      1      ; binary value
      LNIB:        DS      1      ; lower binary nibble
      UNIB:        DS      1      ; upper binary nibble
      CHEXB1:      DS      1      ; call addr upper byte
      CHEXB2:      DS      1      ;  "   "   lower  "

; begin  subroutine

HEXBIN_CODE   SEGMENT CODE
RSEG   HEXBIN_CODE

      HEXBIN:

             POP    CHEXB2       ; save the
             POP    CHEXB1       ;   return address

             POP    LOWER        ; input the
             POP    UPPER        ;   hex value

             PUSH   UPPER        ; find upper binary nibble

             CALL   HEXNIB

             POP    UNIB

             PUSH   LOWER        ; find lower binary nibble

             CALL   HEXNIB

             POP    LNIB
```

```
        ; shift left four bits of upper nibble

    MOV     A,UNIB

    RL      A
    RL      A
    RL      A
    RL      A

    MOV     UNIB,A

        ; combine nibbles to form binary byte

    MOV     A,UNIB

    ADD     A,LNIB

    MOV     BINARY,A

    PUSH    BINARY        ; return the value

    PUSH    CHEXB1        ; load the

    PUSH    CHEXB2        ;   return address

    RET

NIBXEH:
```

```
; subroutine binhex
;
; local definitions and variables
;
; const

; var

BINHEX_DATA   SEGMENT DATA
RSEG   BINHEX_DATA

     OUTHUP:     DS     1      ; upper hex character out
     OUTHLO:     DS     1      ; lower hex character out
      INBIN:     DS     1      ; input binary byte
      UPNIB:     DS     1      ; upper binary nibble
      LONIB:     DS     1      ; lower
      CBINH1:    DS     1      ; call addr upper byte
      CBINH2:    DS     1      ;   "   "   lower   "

; begin subroutine

BINHEX_CODE   SEGMENT CODE
RSEG   BINHEX_CODE

     BINHEX:

            POP    CBINH2     ; save the
            POP    CBINH1     ;  return address

            POP    INBIN

                ; divide inbin into the two nibbles upnib and lonib

                ; lonib first - strip off upper four bits

            MOV    A,INBIN

            ANL    A,#00001111B     ; low four bit mask

            MOV    LONIB,A
```

```
    ; upnib second - strip off lower four bits

MOV    A,INBIN

ANL    A,#11110000B      ; high four bit mask

MOV    UPNIB,A

    ; shift right four bits of upnib to place the bits
    ; in lowest four bit position

MOV    A,UPNIB

RR     A
RR     A
RR     A
RR     A

MOV    UPNIB,A

    ; finish the conversion passing the nibbles to nibhex

PUSH   UPNIB

CALL   NIBHEX

POP    OUTHUP

PUSH   LONIB

CALL   NIBHEX

POP    OUTHLO
```

```
     ; return outhup and outhlo

    PUSH   OUTHLO

    PUSH   OUTHUP

    PUSH   CBINH1        ; load the
    PUSH   CBINH2        ;   return address

    RET

XEHNIB:
```

```
; subroutine hexnib
;
; local definitions and variables
;
; const

    MAXLEN      EQU  0FH   ; maximum length of single
                           ; digit hex numbers

HEXNIB_CODE   SEGMENT CODE
RSEG  HEXNIB_CODE

    HEXTAB:     DB      30H,31H,32H,33H,34H,35H,36H,37H,38H,39H
                DB      41H,42H,43H,44H,45H,46H
                      ; the ascii hexidecimal look-up table

; var

HEXNIB_DATA   SEGMENT DATA
RSEG  HEXNIB_DATA

    HEX:        DS     1      ; hex character
     NIB:       DS     1      ; binary nibble
    LENGTH:     DS     1      ; pointer in hextab
     TABVAL:    DS     1      ; value returned from hextab
     EXIT:      DS     1      ; condition flag for exiting
    CHEXN1:     DS     1      ; call addr upper byte
    CHEXN2:     DS     1      ;  "   "  lower  "

; begin subroutine

RSEG  HEXNIB_CODE

    HEXNIB:

        POP    CHEXN2      ; save the
        POP    CHEXN1      ;   return address

        POP    HEX         ; input hex value
     ; point to beginning of hex table - hextab
    MOV     DPTR,#HEXTAB
      ; initialize length counter
    MOV     LENGTH,#0
      ; clear the exit condition flag
    MOV     EXIT,#FALSE
     ; while (exit = false)
   %MWHILE(EXIT,MXEQY,#FALSE,HNIB1,HNIB2)
            ; look up hex value in table
          CLR     A              ;  no  offset
          MOVC    A,@A +DPTR
          MOV     TABVAL,A
            ; if (hex = tabval) then
         %MIF(HEX,MXEQY,TABVAL,HNIB3)
                   JMP     HNIB2   ; break from the while loop
          ; else
         %MELSE(HNIB3,HNIB4)
               INC    DPTR
               INC    LENGTH
```

```
                ; if (length => 16) then

              %MIF(LENGTH,MXGEY,#MAXLEN,HNIB5)

                   MOV    EXIT,#TRUE

                HNIB5:        ; endif

            HNIB4:        ; endif

        ; end while

        %MWEND(HNIB1,HNIB2)

        ; return

        MOV   NIB,LENGTH

        PUSH   NIB              ; output the nibble

        PUSH   CHEXN1           ; load the
        PUSH   CHEXN2           ;   return address

        RET

BINXEH:
```

```
; subroutine nibhex
;
; local definitions and variables
;
; const

      OFFSET        EQU    37H    ; big ltr ascii hex offset

; var

NIBHEX_DATA   SEGMENT DATA
RSEG   NIBHEX_DATA

     BINNIB:      DS     1       ; binary nibble
    OUTHEX:       DS     1       ; hex character
     CNIBH1:      DS     1       ; call addr upper byte
     CNIBH2:      DS     1       ;  "    " lower   "

; begin subroutine

NIBHEX_CODE  SEGMENT CODE
RSEG   NIBHEX_CODE

    NIBHEX:

          POP    CNIBH2          ; save the
          POP    CNIBH1          ;   return address

          POP    BINNIB          ; input the nibble

            ; if (binnib > 9) then outhex := biga +binnib

         %MIF(BINNIB,MXGTY,#9,NHEX1)

              MOV    A,#OFFSET

              ADD    A,BINNIB

              MOV    OUTHEX,A
          ; else

         %MELSE(NHEX1,NHEX2)

                MOV    A,#NUM0

                ADD    A,BINNIB

               MOV    OUTHEX,A

          NHEX2:   ; end if

           ; return outhex

          PUSH   OUTHEX        ; output the hex value

          PUSH   CNIBH1        ; load the
          PUSH   CNIBH2         ;  return address

          RET

      XEHBIN:



  END
```

APPENDIX G

Source Code Listing of the Executive Program for Controlling the String Bottom Controller (MERLIN)

The source code for program MERLIN consists of 47 modules which are listed below in Table G.1 according to functional category:

Table G.1 Listing Order of MERLIN Routines

Class	Name	Description
Main:		
	main	Main program (MERLIN)
	inpoll	Poll all input ports for input notifying system of any characters received
	cmdpoll	Parse the command string for a valid command - also, handle exceptions
	timpoll	Parse for a STRING command, if found, initiate time out for reply
	allpoll	Parse for an "all call" STRING command - handle exception
	evmpoll	Parse for an environmental module command - handle exception
	nextbyte	Place next byte in command buffer
	cmdserv	Service a complete command, else call error routine
	allserv	Exception code for an "all call" command
	evmserv	Exception code for an environmental module command

Commands:

coinpat	Coincidence pattern
rawsel	Raw select
stkhgt	Stack height
source	Source on/off
sorsel	Source select
xtrbit	Extra bit (drive spare lines)
readad	Read an analog (A to D) channel
mreset	Master reset

I/O Routines:

datlat	Move SBC optical path data from internal table to external latch
tablat	Move SBC latch control data from internal table to external latch
outlatch	Output a byte to a latch port
inport	Read a byte from specified port
outport	Write a byte to specified port

Error Routines:

deverr	Device error reply
forerr	Format error reply
cmderr	Command error reply
timerr	String timeout error reply

Communications:

msgreply	Output message on CAB and OPT
allecho	Output byte on CAB, STR, PWR, & OPT
upecho	Output byte on CAB and OPT
downecho	Output byte on PWR and STR
evmmsg	Output fake environmental module message on CAB and OPT
allmsg	Output fake "all call" replies on CAB and OPT

Type Checking and Conversion:

ascbin	Convert ASCII character to binary
binhex	Convert byte sized binary numbers to two byte hex equivalent
nibhex	Convert four bit binary nibble into ASCII hex character
ishex	Test character if it's an ASCII hex number

Initialization and Control:

init	Initialize CAB and STR at 300 baud
deinit	Return to SB-180 ROM monitor
delay	Delay one millisecond

Device Drivers:

atod	Convert specified channel analog input to digital value
incab	Input a character from CAB
instr	Input a character from STR
inpwr	Input a character from PWR
outcab	Output a character to CAB
outstr	Output a character to STR
outpwr	Output a character to PWR
outopt	Output a character to OPT

Header (include) Files:

SBCDEF.H	Common SBC system definitions
SBCGLOB.H	Declarations for global variables, mostly for debugging
SBCMON.H	Common ROM monitor definitions

The listing of the individual modules now follows:

```
/*
 * MAIN - Main driver for the String Bottom Controller
 * program MERLIN.
 *
 *     This sets 'sbcrun' flag to TRUE before entering the
 *  top level loop, indicating normally running the SBC
 *  program. If a command from <CABLE> is [ZZ], 'sbcrun' flag
 *  will be set to FALSE in 'CMDSERV' which is called by
 *  'CMDPOLL', then exits from the top level loop and returns
 *  to the ROM monitor.
 *
 *     (Note: This calls temporary routine 'CLEAR' to
 * initialize all global variables.  Initial values of
 * table[] have to be defined and assigned in 'INIT' later.)
 *
 *     Routines called: INIT, INPOLL, CMDPOLL, UPECHO, DEINIT
 *
 * authors John F. Babson and Yoshiko Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date: Jan. 26, 1987
 *
 */


#include SBCDEF.H

/* declare all global variables here */

#include SBCGLOB.H

main() /* control the SBC modules */
{

     /* initialize the SBC program (interrupt, baud rate, */
     /* default parameters for the SBC latch)        */

     INIT();

     CLEAR();   /* temporary initialization of globals */
```

```
/* set flag to normal loop condition */

sbcrun = TRUE;

/* repeat until get a command [ZZ] from <CABLE> */

do {
     /* poll input ports <CAB>, <STR> and <PWR> */
     /* to fetch a character                   */

     INPOLL();

     if (inflg & ONCAB) {       /* inflg(CAB) is on */

          /* poll for a command and parse it  */
          /* if complete, execute each command      */

          CMDPOLL();
     }

     else if (inflg & ONSTR) { /* inflg(STR) is on */

          /* echo a character from <STR>
             back to <CAB> */

          UPECHO(inbuf[STR]);

          inflg = inflg & OFFSTR;/* clear inflg(STR) */
     }

     else if (inflg & ONPWR) {  /* inflg(PWR) is on */

          /* echo a character from <PWR>
             back to <CAB> */

          UPECHO(inbuf[PWR]);

          inflg = inflg & OFFPWR;/* clear inflg(PWR) */
     }

} while (sbcrun == TRUE);
```

```
     /* reset the SBC program (error flag, interrupt) */
     /* and return to the SBC monitor          */

     DEINIT();

} /* end of main */


CLEAR() /* temporary initialization */
{

     inflg = 0;
     dev = 0;
     bytnum = 0;

     poll = FALSE;/* disable exception parsing for EVMPOLL */
     talk = FALSE;/* disable exception parsing for ALLPOLL */
     time = FALSE;/* disable exception parsing for TIMPOLL */
     timout = FALSE; /* disable software STRING / POWER time
                        out counter */
     count = MAXCNT; /* set count down time out counter to
                        maximum */


     inbuf[0] = 0; inbuf[1] = 0; inbuf[2] = 0;

     cmd[0] = 0; cmd[1] = 0; cmd[2] = 0; cmd[3] = 0;

     table[0] = 0; table[1] = 0; table[2] = 0;
     table[3] = 0; table[4] = 0; table[5] = 0;
     table[6] = 0; table[7] = 0; table[8] = 0;

} /* end of clear */
```

```
/*
 * INPOLL - Routine to poll all input ports to fetch
 * available characters. These characters are placed into the
 * buffer 'INBUF' and a flag is set in 'INFLG' so that they
 * may be grabed at a latter date by other routines in an
 * asynchronous manner.
 *
 *       Routines called : INCAB, ALLECHO, INSTR, INPWR
 *
 * author - John Babson,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : Jan. 23, 1987
 *
 */


#include SBCDEF.H

    extern int inbuf[];
    extern int inflg;
    extern int time;
    extern int timout;
    extern int count;
    extern int bytnum;

INPOLL() /* poll all input ports for a character */
{

    int c; /* input charecter */


     /* first, look for input from the cable */

     c = incab();

     if (c != NULL) {

          inbuf[CAB] = c;
```

```
        /* set bit inflg(CAB) */

        inflg = (inflg & OFFCAB) | ONCAB;

        allecho(c);  /* echo the input back upstairs */

}


/* second, look for a character from the string */

c = instr();

if (c != NULL) {

        inbuf[STR] = c;

        /* set bit inflg(STR) */

        inflg = (inflg & OFFSTR) | ONSTR;

        /* with an input character, reset software
           time out */

        time = FALSE;   /* clear exception parsing flag */
        count = MAXCNT; /* reaset count down variable */

}


/* last, look for a character from the power module */

c = inpwr();

if (c != NULL) {

        inbuf[PWR] = c;
        /* set bit inflg(PWR) */

        inflg = (inflg & OFFPWR) | ONPWR;

        /* with an input character, reset software
           time out */

        time = FALSE;   /* clear exception parsing flag */
        count = MAXCNT; /* reset count down variable */

    }

    /* finally, before completing the character input poll,
       check to see if a software time out is in process
       looking for input from either the STRING or POWER
       ports */

    if (timout == TRUE) { /* software time out test */
        count--;

        if (count == 0) {    /* time out message */
            TIMERR();
            time= FALSE;     /* clear exception
                                parsing flag */
            count = MAXCNT;      /* reset count down
                                    variable */
            bytnum = 0;
        }
    }

} /* end of inpoll */
```

```
/*
 *
 * CMDPOLL - Routine to grab a next character, checking
 * device error & format error.  Also trap exception
 * conditions of EVM and STRING "all call" commands.  If a
 * complete command is is received, execute it by calling
 * 'CMDSERV'.
 *
 *        Routines called : DEVERR, NEXTBYTE, CMDSERV,
 *        FORERR, ISHEX, EVMPOLL, ALLPOLL, TIMPOLL
 *
 * authors  John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : Jan. 23, 1987
 *
 */


#include SBCDEF.H

    extern int      inbuf[];
    extern int      inflg;
    extern int      dev;
    extern int      bytnum;
    extern int      poll;
    extern int      talk;
    extern int      time;
    extern int      timout;
    extern int      count;


CMDPOLL()      /* get a next char, checking error, and
            if complete command, execute it       */
{
    int c; /* data character from CABLE */



     c = inbuf[CAB];
     inflg = inflg & OFFCAB;          /* clear inflg(CAB) */
```

```
/* first, check flags to see if an exception parser is
   needed */

if (poll == TRUE) {


     EVMPOLL(c);      /* environmental module poll */
     return;

}
if (talk == TRUE) {


     ALLPOLL(c);      /* all call to STRING */
     return;

}
if (time == TRUE) {


     TIMPOLL();       /* time out routine for STRING /
                         POWER */
     return;

/* no exception parser yet needed so continue the
   parsing here */

} else {
     if (bytnum == 0) {

               if (c == DOL) {
                    bytnum = 1;


               talk = FALSE; /* reset talk switch */

          }
               else {
                    bytnum = 0;
          }
      }
      else if (bytnum == 1) {

                  if (c == DOL) {
                        bytnum = 1;
            }
                  else {
                        bytnum = 2;
                        dev = c;
                  }
      }
      else if (bytnum == 2) {

                  if (c == DOL) {
                        bytnum = 1;
            }
                  else if ((c == dev) && (ISHEX(c) ==
                              TRUE)) {
                        bytnum = 3;

                  /* device is valid so check for exception
                     parsing */

                  if (dev == EVM)        { /* dev is env.
                                              module */

                        /* force exception parsing of
                           command by EVMPOLL starting
                           on next pass */


                        poll = TRUE;
```

```
} else if (dev == ALLMOD) {/*dev is all*/

    /* force exception parsing of
       command by ALLPOLL starting
       on next pass */


    talk = TRUE;     /* modules call */

} else if (dev == ALLOM) { /*dev is all*/

    /* force exception parsing of
       command by ALLPOLL starting
       on next pass */


    talk = TRUE;     /* OMs call    */

} else if (dev == ALLCM)  {/*dev is all*/

    /* force exception parsing of
       command by ALLPOLL starting
       on next pass */


    talk = TRUE;     /* CMs call    */

} else if ((dev != LATCH) &&
           (poll != TRUE) &&
           (talk != TRUE)) {

       /* force exception parsing of
       command by TIMPOLL starting
       on next pass */


    time = TRUE;     /* STRING/POWER */

}
```

```
                else {
                     poll = FALSE; /* ? still needed ? */

                }
           }
                     else { /* invalid device so issue
                               error message and start
                               over */

                bytnum = 0;
                     DEVERR();
                return;
           }

      /* continue regular SBC command parsing eventually
         executing the command (CMDSERV) or issuing a
         format error (FORERR) */

      }
      else if (bytnum == 3) {

                NEXTBYTE(c);          /* cmd[0] */
      }
      else if (bytnum == 4) {

                NEXTBYTE(c);          /* cmd[1] */
      }
      else if (bytnum == 5) {

                NEXTBYTE(c);          /* cmd[2] */

                if (c == CR) {
                     CMDSERV();       /* complete a
                                         command */
                bytnum = 0;
                return;

           }
      }
```

```
            else if (bytnum == 6) {

                    NEXTBYTE(c);            /* cmd[3] */
            }
            else if (bytnum == 7) {

                    if (c == CR) {
                        CMDSERV();          /* complete a
                                               command */
                    bytnum = 0;
                }
                    else {
                        bytnum = 0;
                        FORERR();
                    }
                return;
            }
            else {
                    bytnum = 0;
                    FORERR();
            }
        }

} /* end of CMDPOLL */
```

```
/*
 *
 * TIMPOLL - Routine to poll a command for the STRING. If a
 * complete command is received, enable time out counter to
 * generate a TIMERR if no response is received within the
 * time out period.
 *
 *        Routines called : NEXTBYTE
 *
 * author John F. Babson,
 * University of Hawaii, Physics
 * Revision date : Jan. 28, 1987
 *
 */


#include SBCDEF.H

    extern int  bytnum;
    extern int  time;
    extern int  timout;

TIMPOLL(c) /* STRING time out poll */
int c;     /* data character from CABLE */
{


     if (bytnum == 3) {

          NEXTBYTE(c);          /* cmd[0] */
     }
     else if (bytnum == 4) {

          NEXTBYTE(c);          /* cmd[1] */

     }
     else if (bytnum == 5) {

          NEXTBYTE(c);          /* cmd[2] */
          if (c == CR) {
               bytnum = 0; /* complete command received */
               time = FALSE;  /* disable exception parsing */
               timout = TRUE; /* enable software time out
                                   in case no input is
                                   received from STRING or
                                   POWER ports */
          }
     }
     else if (bytnum == 6) {

          NEXTBYTE(c);           /* cmd[3] */
     }
     else if (bytnum == 7) {

          NEXTBYTE(c);           /* cmd[4] */

          if (c == CR) {
               bytnum = 0;  /* complete command received */
               time = FALSE;  /* disable exception parsing */
               timout = TRUE; /* enable software time out
                                   in case no input is
                                   received from STRING or
                                   POWER ports */
          }
     }
     else {

          bytnum = 0;

     }
} /* end of TIMPOLL */
```

```
/*
 *
 * ALLPOLL - Routine to poll a command for all calls to the
 * STRING.
 *    If a complete command is received, execute it by
 *    calling 'ALLSERV'.
 *
 *          Routines called : NEXTBYTE, ALLSERV, FORERR
 *
 * author John F. Babson,
 * University of Hawaii, Physics
 * Revision date : Jan. 23, 1987
 *
 */

#include SBCDEF.H

    extern int  bytnum;
    extern int  talk;


ALLPOLL(c)  /* STRING "all call" exception code */
int c;      /* data character from CABLE */
{


     if (bytnum == 3) {

          NEXTBYTE(c);          /* cmd[0] */

     }
     else if (bytnum == 4) {

          NEXTBYTE(c);          /* cmd[1] */

     }
     else if (bytnum == 5) {

          NEXTBYTE(c);          /* cmd[2] */

     }
    else if (bytnum == 6) {

         NEXTBYTE(c);          /* cmd[3] */

    }
    else if (bytnum == 7) {

         NEXTBYTE(c);          /* cmd[4] */

         if (c == CR) {
              ALLSERV();       /* complete the all call
                                command */
              bytnum = 0;
              talk = FALSE;
         }
         else {
              bytnum = 0;
              talk = FALSE;
              FORERR();
         }
    }
    else {
         bytnum = 0;
         talk = FALSE;
         FORERR();

    }

} /* end of ALLPOLL */
```

```
/*
 *
 * EVMPOLL - Routine to poll a command for the environmental
 * module.
 *        If a complete command is received, execute it by
 *        calling 'EVMSERV'.
 *
 *        Routines called : NEXTBYTE, EVMSERV, FORERR
 *
 * authors Y. Miyakoshi and John F. Babson,
 * Univerisity of Hawaii
 * Revision by John F. Babson
 * Revision date : Jan. 23, 1987
 *
 */

#include SBCDEF.H

    extern int  bytnum;
    extern int  poll;

EVMPOLL(c) /* environmental module poll */
int c;     /* data character from CABLE */
{


      if (bytnum == 3) {

            NEXTBYTE(c);          /* cmd[0] */
      }
      else if (bytnum == 4) {

            NEXTBYTE(c);          /* cmd[1] */

            if (c == CR) {
                  EVMSERV(); /* case 1 - complete a single */
                  bytnum = 0;     /* letter command */
                  poll = FALSE;
            }

      }
```

```
else if (bytnum == 5) {

     NEXTBYTE(c);           /* cmd[2] */
}
else if (bytnum == 6) {

     NEXTBYTE(c);           /* cmd[3] */
}
else if (bytnum == 7) {

     NEXTBYTE(c);           /* cmd[4] */
}
else if (bytnum == 8) {

     NEXTBYTE(c);           /* cmd[5] */

     if (c == CR) {
          EVMSERV();        /* case 2 - complete an */
          bytnum = 0;       /* F command */
          poll = FALSE;
     }
}
else if (bytnum == 9) {

     NEXTBYTE(c);           /* cmd[6] */
}
else if (bytnum ==10) {

     NEXTBYTE(c);           /* cmd[7] */

     if (c == CR) {
          EVMSERV();        /* case 3 - complete a */
          bytnum = 0;       /* D command */
          poll = FALSE;
     }
     else {
          bytnum = 0;
          poll = FALSE;
          FORERR();
     }
}
```

```
        else {
              bytnum = 0;
              poll = FALSE;
              FORERR();
        }

  } /* end of EVMPOLL */
```

```
/*
 *
 * NEXTBYTE - Routine to get a byte into CMD0, CMD1, ...
 * CMD7, checking format error.
 *
 *              Routines called : FORERR
 *
 * authors Y. Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : Jan. 28, 1987
 *
 */


#include SBCDEF.H

    extern int  cmd[];
    extern int  bytnum;
    extern int  poll;
    extern int  talk;
    extern int  time;
    extern int  timout;
    extern int  count;

NEXTBYTE(byte) /* get a byte into CMD0, CMD1, ..., CMD7 */
int byte;
{
    int i;


        if (byte == DOL) {

                bytnum = 1;

                poll = FALSE;    /* disable exception parsing
                                 for EVMPOLL */
                talk = FALSE;    /* disable exception parsing
                                 for ALLPOLL */
                time = FALSE;    /* disable exception parsing
                                 for TIMPOLL */
                timout = FALSE; /* disable software STRING /
                                POWER time out counter */
                count = MAXCNT; /* set count time out counter
                                to maximum */

                FORERR();
        }
        else {

                i = bytnum - 3;
                cmd[i] = byte;
                bytnum++;
        }

} /* end of NEXTBYTE */
```

```
/*
 *
 * CMDSERV - Routine to do command services or return an
 * error message for invalid commands.
 *
 *         If a command is [ZZ], set normal loop condition to
 *         FALSE to return SB180 Monitor.
 *
 *         Command format :
 *            $ <DEV> <DEV> <CMD0> <CMD1> <CR>
 *         or $ <DEV> <DEV> <CMD0> <CMD1> <DATA0> <DATA1><CR>
 *
 *         Routines called : COINPAT, RAWSEL, STKHGT, SOURCE,
 *                           SORSEL, XTRBIT, READAD, MRESET,
 *                           ALLSERV, CMDERR
 *
 * authors John F. Babson and Y. Miyakoshi
 * Revision date : Nov. 17, 1986
 * Revised by Y. Miyakoshi
 *
 */


#include SBCDEF.H

    extern int  dev;
    extern int  cmd[];
    extern int  sbcrun;


CMDSERV()       /* service the command or return an error
                   message for invalid commands            */
{


     if (dev != LATCH)
          return;
```

```
if (cmd[0] == 'C') {
          if (cmd[1] == 'P') {
               if (cmd[2] == '0')
                    COINPAT(); /* command is [CP] */
               else
                    CMDERR();
          }
          else
               CMDERR();
}
else if (cmd[0] == 'R') {
          if (cmd[1] == 'S') {
               if (cmd[2] == '0')
                    RAWSEL();/* command is [RS] */
               else
                    CMDERR();
     }
     else
          CMDERR();
}
else if (cmd[0] == 'S') {
          if (cmd[1] == 'H') {
               if (cmd[2] == '0')
                    STKHGT();/* command is [SH] */
               else
                    CMDERR();
               }
          else if (cmd[1] == 'O')
                  SOURCE();      /* command is [SO] */

          else if (cmd[1] == 'S')
                  SORSEL();   /* command is [SS] */
          else
               CMDERR();
}
else if (cmd[0] == 'X') {
          if (cmd[1] == 'B')
               XTRBIT();  /* command is [XB] */
          else
               CMDERR();
}
```

```
    else if (cmd[0] == 'A') {
            if (cmd[1] == 'D') {
                 if (cmd[2] == '0')
                      READAD(); /* command is [AD] */
                 else
                      CMDERR();
            }
            else
                 CMDERR();
    }
    else if (cmd[0] == 'M') {
            if (cmd[1] == 'R')
                 MRESET();         /* command is [MR] */
            else
                 CMDERR();
    }
    else if (cmd[0] == 'Z') {
            if (cmd[1] == 'Z')
                 sbcrun = FALSE; /* command is [ZZ] */
            else
                 CMDERR();
    }
    else if (cmd[0] == 'W') {
            if (cmd[1] == 'E' || cmd[1] == 'F')
                 ALLSERV();        /* command is [WE/WF] */
            else
                 CMDERR();
    }
    else
         CMDERR();

} /* end of CMDSERV */
```

```
/*
 *
 * ALLSERV - Parser for O.M. and C.M. commande ALL-CALL.
 *         Parallels EVMSERV.
 *
 *         Command format : W <E/F> <0-F> <0-F>
 *
 *         Routines called : ALLMSG, CMDERR
 *
 * author John F. Babson,
 * University of Hawaii, Physics
 * Revised by Yoshiko Miyakoshi
 * Revision date: Dec. 4, 1986
 *
 */


#include SBCDEF.H

    extern int cmd[];

ALLSERV() /* string "all call" service */
{

     if (cmd[0] == 'W') {
          if ((cmd[1] == 'E') || (cmd[1] == 'F')) {
               if (ISHEX(cmd[2]) == TRUE &&
                   ISHEX(cmd[3]) == TRUE)

                    ALLMSG();

               else

                    CMDERR();
          }
     }

} /* end of ALLSERV */
```

```
/*
 *
 * EVMSERV - Routine to do command service for the
 * environmental module or return an error message for
 * invalid command.
 *
 *        Command format :
 *           $ B B {R,Z,T,L,B,O} <CR>
 *        or $ B B F HEX(0) HEX(1) HEX(2) HEX(3) <CR>
 *        or $ B B D HEX(0) HEX(1) HEX(2) HEX(3) 1 1 <CR>
 *
 *        Routines called : EVMMSG, EVMCMD, CMDERR, ISHEX
 *
 * authors Y. Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : Dec. 4, 1986
 *
 */


#include SBCDEF.H

   extern int       cmd[];
   extern char command[];     /* for debugging */


EVMSERV() /* environmental module service */
{

     command[0] = 'E'; command[1] = 'M';


     /* case 1 : {R,Z,T,L,B,O} */

     if (cmd[0] == 'R' || cmd[0] == 'Z' || cmd[0] == 'T' ||
         cmd[0] == 'L' || cmd[0] == 'B' || cmd[0] == 'O'   )
     {

          EVMMSG();
}

/* case 2 : F HEX(0) HEX(1) HEX(2) HEX(3) */

else if (cmd[0] == 'F') {
          if    (ISHEX(cmd[1]) == TRUE &&
                 ISHEX(cmd[2]) == TRUE &&
                 ISHEX(cmd[3]) == TRUE &&
                 ISHEX(cmd[4]) == TRUE    )

                 EVMMSG();
          else
                 CMDERR();
     }

/* case 3 : D HEX(0) HEX(1) HEX(2) HEX(3) 1 1 */

else if (cmd[0] == 'D') {
          if  (ISHEX(cmd[1]) == TRUE &&
                 ISHEX(cmd[2]) == TRUE &&
                 ISHEX(cmd[3]) == TRUE &&
                 ISHEX(cmd[4]) == TRUE    ) {

                 if (cmd[5] == '1' && cmd[6] == '1') {

                 /* do nothing - this is the EM 300
                    baud dump command so no second
                    echo reply is expected */

                 }
                 else
                      CMDERR();
          }
          else
                 CMDERR();
     }
     else
          CMDERR();


} /* end of EVMSERV */
```

```
/*
 *
 * COINPAT - Routine for the command service 'coincidence
 * pattern'.
 *
 *            Command format : C P <0> <0-F>
 *
 *         Routines called : ISHEX, ASCBIN, ALLMSG, TABLAT,
 *                           CMDERR
 *
 *
 * authors John F. Babson and  Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : Nov 21, 1986
 *
 */


#include SBCDEF.H

    extern int  cmd[];
    extern char table[];
    extern char command[];     /* for debugging */


COINPAT() /* command service - coincidence pattern -
              C P <0> <0-F> */
{


     command[0] = 'C'; command[1] = 'P';
    if (ISHEX(cmd[3]) == TRUE) {

         ALLMSG();

         table[5] = (table[5] & 0xF0) | (ASCBIN(cmd[3]));
                                   /* write onto LH of
                                      table[5] */

         TABLAT();
    }
    else
         CMDERR();

} /* end of COINPAT */
```

```
/*
 *
 * RAWSEL - Routine for the command service 'raw select'.
 *
 *        Command format : R S <0> <0-F>
 *
 *        Routines called : ISHEX, ASCBIN, ALLMSG, TABLAT,
 *                          CMDERR
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : Nov 21, 1986
 *
 */

#include SBCDEF.H

    extern int  cmd[];
    extern char table[];
    extern char command[];      /* for debugging */

RAWSEL() /* command service - raw select - R S <0> <0-F> */
{
     command[0] = 'R'; command[1] = 'S';


     if (ISHEX(cmd[3]) == TRUE) {

          ALLMSG();

          table[5] = (table[5] & 0x0F) |
                     ((ASCBIN(cmd[3])) << 4);
                     /* write onto UH of table[5] */

          TABLAT();
     }
     else
          CMDERR();

} /* end of RAWSEL */
```

```
/*
 *
 * STKHGT - Routine for the command service 'stack hight'.
 *
 *        Command format : S H <0> <0-F>
 *
 *        Routines called : ISHEX, ASCBIN, ALLMSG, TABLAT,
 *                          CMDERR
 *
 * authors John F. Babson and  Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : Nov 21, 1986
 */

#include SBCDEF.H

    extern int  cmd[];
    extern char table[];
    extern char command[];      /* for debugging */


STKHGT() /* command service - stack hight - S H <0> <0-F> */
{
     command[0] = 'S'; command[1] = 'H';

     if (ISHEX(cmd[3]) == TRUE) {

          ALLMSG();

          table[4] = (table[4] & 0x0F) |
                         ((ASCBIN(cmd[3])) << 4);
          /* write onto UH of table[4] */

          TABLAT();
     }
     else
          CMDERR();

} /* end of STKHGT */
```

```
/*
 *
 * SOURCE - Routine for the command service 'source on/off'.
 *
 *           Command format : S O <0-B> <0/1>
 *
 *        Routines called : ASCBIN, ALLMSG, TABLAT, CMDERR
 *
 * authors John F. Babson and Miyakoshi,
 * University of Hawaii, Physics
 * revised by Miyakoshi
 * Revision date : Nov 21, 1986
 *
 */


   extern int  cmd[];
   extern char table[];
   extern char command[];     /* for debugging */


SOURCE() /*command service - source on/off -
           S O <0-B> <0/1> */
{
   char *ptr;
   int cmd2;
   int cmd3;
   int i;
   int bit;


    command[0] = 'S'; command[1] = 'O';
    if ( ((cmd[2] >= '0' && cmd[2] <= '9') ||
           cmd[2] == 'A' || cmd[2] == 'B') &&
          (cmd[3] == '0' || cmd[3] == '1') )

    {

         ALLMSG();

         cmd2 = ASCBIN(cmd[2]);
         cmd3 = ASCBIN(cmd[3]);

         ptr = table + 3;      /* ptr points to table[3] */

         if (cmd2 > 7) {
                   cmd2 = cmd2 - 8; /* set bit position
                                       to 0 */
                   ptr++;           /* go to next byte
                                       location */
         }

         bit = 1;

         if (cmd2)
                   for (i = 0; i < cmd2; i++)
                        bit = bit << 1;

         *ptr = *ptr & ~bit;        /* reset one bit and */
                        /* mask others         */
         if (cmd3)
                   *ptr = *ptr | bit;    /* set one bit */

         TABLAT();
    }
    else
         CMDERR();

} /* end of SOURCE */
```

```
/*
 *
 * SORSEL - Routine for the command service 'source select'.
 *
 *       Command format : S S <1-9> <0-3>
 *
 *       Routines called : ASCBIN, ALLMSG, TABLAT, CMDERR
 *
 * authors John F. babson and  Y. Miyakoshi,
 * University of Hawaii
 * revised by Y. Miyakoshi
 * Revision date : Nov 21, 1986
 *
 */


   extern int  cmd[];
   extern char table[];
   extern char command[];     /* for debugging */


SORSEL() /* command service - source select -
             S S <1-9> <0-3> */
{
   char *ptr;
   int cmd2;
   int cmd3;


    command[0] = 'S'; command[1] = 'S';


    if ( (cmd[2] >= '1' && cmd[2] <= '9') &&
         (cmd[3] >= '0' && cmd[3] <= '3')    ) {

         ALLMSG();

         cmd2 = ASCBIN(cmd[2]);
         cmd3 = ASCBIN(cmd[3]);

         ptr = table + 6; /* ptr points to table[6] */
```

```
          if (cmd2 >= 5 && cmd2 <= 8)
               ptr++;     /* ptr points to table[7] */
          else if (cmd2 == 9)
               ptr = ptr + 2;/* ptr points to table[8]*/


     if (cmd2 == 2 || cmd2 == 6)
                    cmd3 = cmd3 << 2;

     else if (cmd2 == 3 || cmd2 == 7)
                    cmd3 = cmd3 << 4;

     else if (cmd2 == 4 || cmd2 == 8)
                    cmd3 = cmd3 << 6;


     if (cmd2 == 1 || cmd2 == 5 || cmd2 == 9)
          *ptr = (*ptr & 0xFC) | cmd3;

     else if (cmd2 == 2 || cmd2 == 6)
          *ptr = (*ptr & 0xF3) | cmd3;

     else if (cmd2 == 3 || cmd2 == 7)
          *ptr = (*ptr & 0xCF) | cmd3;

     else if (cmd2 == 4 || cmd2 == 8)
          *ptr = (*ptr & 0x3F) | cmd3;


          TABLAT();
  }
  else
     CMDERR();

} /* end of SORSEL */
```

```
/*
 *
 * XTRBIT - Routine for the command service 'extra bit'.
 *
 *        Command format : X B <0-5> <0/1>
 *
 *        Routines called : ASCBIN, ALLMSG, TABLAT, CMDERR
 *
 * authors John F. Babson and  Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : Nov 21, 1986
 *
 */


   extern int  cmd[];
   extern char table[];
   extern char command[];      /* for debugging */


XTRBIT() /* command service - extra bit - X B <0-5> <0/1> */
{
   int cmd2;
   int cmd3;
   int i;
   int bit;


     command[0] = 'X'; command[1] = 'B';


     if ( (cmd[2] >= '0' && cmd[2] <= '5') &&
          (cmd[3] == '0' || cmd[3] == '1')     ) {

          ALLMSG();

          cmd2 = ASCBIN(cmd[2]);
          cmd3 = ASCBIN(cmd[3]);

          bit = 4;
          if (cmd2)
                    for (i = 0; i < cmd2; i++)
                         bit = bit << 1;

          table[8] = table[8] & ~bit; /* reset one bit and
                                         mask others */
          if (cmd3)
                    table[8] = table[8] | bit;/*set one bit*/

          TABLAT();
     }
     else
          CMDERR();

} /* end of XTRBIT */
```

```
/*
 *
 * READAD - Routine for the command service 'atod
 * conversion'.
 *
 *       Command format : A D <0> <0-F>
 *
 *       Routines called : ISHEX, ASCBIN, ATOD, BINHEX,
 *                         UPECHO, CMDERR
 *
 * authors Y. Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : Feb. 23, 1987
 *
 */


#include SBCDEF.H

    extern int  cmd[];
    extern char command[];      /* for debugging */


READAD() /* command service - ATOD conversion - A D <0> <0-F>
            make sure message is only ROM code */

{

    char value;       /* returned value from ATOD */
    char upbyte;
    char lobyte;
    int  chan;        /* channel # for ATOD */


     command[0] = 'A'; command [1] = 'D';

     if (ISHEX(cmd[3]) == TRUE) {

          chan = ASCBIN(cmd[3]);
```

```
        value = ATOD(chan);

        BINHEX(value, &upbyte, &lobyte);

        UPECHO(DOL);
        UPECHO('C');
        UPECHO('C');
        UPECHO('A');
        UPECHO('D');

        UPECHO(upbyte);
        UPECHO(lobyte);
        UPECHO(CR);
    }
    else
        CMDERR();

} /* end of readad */
```

```
/*
 *
 * MRESET - Routine for the command service 'master reset'.
 *
 *        Command format : M R
 *
 *        Routines called : OUTLATCH
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 28, 1986
 *
 */


#include SBCDEF.H

    extern int  cmd[];
    extern char command[];      /* for debugging */


MRESET() /* master reset */
{

     command[0] = 'M'; command[1] = 'R';

     outlatch(MASTST, 0);       /* strobe the master reset */


} /* end of mreset */
```

```
/*
 *
 * DATLAT - Routine to move the SBC optical path data from
 * the internal table into the external latch.
 *
 *        Routines called : OUTLATCH
 *
 * authors - John Babson and Dave Harris,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 23, 1986
 *
 */


#include SBCDEF.H

    extern char table[];


int datlat() {

    int i;

     for (i = 0; i < 3; i++) { /* move the data into the
                                   latch */

         outlatch(LATCH0 + i, table[i]);
     }

     outlatch(DATAST,0);  /* strobe the data */


} /* end of datlat */
```

```
/*
 *
 * TABLAT - Routine to move the SBC optical path data from
 * the internal table into the external latch.
 *
 *           Routines called : OUTLATCH
 *
 * authors - John Babson and Dave Harris,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 19, 1986
 *
 */


#include SBCDEF.H

    extern char table[];

int tablat() {

        int i;

        for (i = 3; i < 9; i++) {

                outlatch(LATCH0 + i, table[i]); /* move the
                                 settings into the latch */
        }

} /* end of tablat */
```

```
/*
 *
 * OUTLATCH - Routine to output  a byte to a specified port.
 *
 *          Routines called :
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : June 2, 1986
 *
 *
 */


outlatch(port, byte) /* write a byte to a specified port */
{

     #asm

          POP  D     ; return address
          POP  H     ; data in HL, H=0
          POP  B     ; port# in BC, B=0

          MOV  A,L         ; move data
          DB   0EDH,79H    ; Z80 OUT (C),A

          PUSH B     ; restore
          PUSH H     ;   stack
          PUSH D     ; restore return address


     #endasm

} /* end of outlatch */
```

```
/*
 *
 * INPORT - Routine to call one of three input routines
 * 'INCAB' for the optical cable, 'INSTR' for the string, or
 * 'INPWR' for  the power switch.
 *
 *       Routines called : INCAB, INSTR, INPWR
 *
 * author - John Babson,
 * University of Hawaii Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 23, 1986
 *
 */


#include SBCDEF.H


int inport(port)
int port;
{

    int c;

     if (port == CAB)
              c = incab();

     else if (port == STR)
              c = instr();

     else if (port == PWR)
              c = inpwr();


     return (c);

} /* end of inport */
```

```
/*
 *
 * OUTPORT - Routine to call one of four output routines.
 *         'outcab' for the optical cable
 *         'outstr' for the string
 *         'outpwr' for the power switch
 *         'outopt' for the SBC optical path
 *
 *         Routines called : OUTCAB, OUTSTR, OUTPWR, OUTOPT
 *
 * authors - John Babson and Dave Harris,
 * University of Hawaii Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 23, 1986
 *
 */


#include SBCDEF.H


outport(port, byte)
int port;
int byte;
{


     if (port == CAB)
               outcab(byte);

     else if (port == STR)
               outstr(byte);

     else if (port == PWR)
               outpwr(byte);

     else
          outopt(byte);


} /* end of outport */
```

```
/*
 *
 * DEVERR - Routine to respond to device error condition.
 *        It sends up an error message to <CAB> & <OPT>, and
 *        also sends down a dollar sign to <STR> & <PWR>.
 *
 *        Message format : $ ? ? D E V R <CR>
 *
 *        Routines called : UPECHO, DOWNECHO
 *
 * authors John F. Babson and  Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John Babson
 * Revision date : Feb. 23, 1986
 *
 */


#include SBCDEF.H


DEVERR()        /* respond to device error condition by sending
                   back error message with '?' -- make sure
                   message is only ROM code. */

{

     UPECHO(DOL);
     UPECHO('?');
     UPECHO('?');

     UPECHO('D');
     UPECHO('E');
     UPECHO('V');
     UPECHO('R');
     UPECHO(CR);

     DOWNECHO(DOL);

} /* end of DEVERR */
```

```
/*
 * FORERR - Routine to respond to message format error.
 *         It sends up an error message to <CAB> & <OPT>, and
 *         also sends down a dollar sigh to <STR> & <PWR>.
 *
 *         Message format : $ ? ? <DEV> <DEV> F O R M <CR>
 *
 *         Routines called : UPECHO, DOWNECHO
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John Babson
 * Revision date : Feb. 23, 1987
 */

#include SBCDEF.H

    extern int  dev;

FORERR()        /* respond to message format error condition by
                   sending back error message with '?' -- make
                   sure message is only ROM code. */
{

      UPECHO(DOL);
      UPECHO('?');
      UPECHO('?');

      UPECHO(dev);
      UPECHO(dev);

      UPECHO('F');
      UPECHO('O');
      UPECHO('R');
      UPECHO('M');
      UPECHO(CR);

      DOWNECHO(DOL);


} /* end of FORERR */
```

```
/*
 *
 * CMDERR - Routine to respond to command error condition.
 *        It sends up the improper command with '?' to <CAB> &
 *        <OPT>, and also sends down a dollar sign to <STR> &
 *        <PWR>.
 *
 *        Message format : $ ? ? <DEV> <DEV> <CMD0> ...
 *                             ... <CMD6> <CR>
 *
 *        Routines called : UPECHO, DOWNECHO
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John Babson
 * Revision date : Feb. 23, 1987
 *
 */


#include SBCDEF.H

    extern int  dev;
    extern int  bytnum;
    extern int  cmd[];


CMDERR()        /* respond to command error condition by
                   echoing back improper command with '?' --
                   make sure message is only ROM code. */
{

    int i;

     UPECHO(DOL);
     UPECHO('?');
     UPECHO('?');

     UPECHO(dev);
     UPECHO(dev);
	if (dev == EVM) {

		for (i = 0; i < bytnum - 3; i++)
			UPECHO(cmd[i]);
	}
	else {

		UPECHO(cmd[0]);
		UPECHO(cmd[1]);
	}

	UPECHO(CR);

	DOWNECHO(DOL);

} /* end of CMDERR */
```

```
/*
 *
 * TIMERR - Routine to send back a time out error message to
 * the CABLE, when no response received from a stting module
 * (i.e. no '$' received within one second or so.)
 *
 *       Message format : $ T I M E <CR>
 *
 *       Routines called : UPECHO
 *
 * authors  Y. Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : DEC. 4, 1986
 *
 */


#include SBCDEF.H

    extern int dev;


timerr() /* time out error message */
      /* routine to send a time out error message back up
         the CABLE */
{
     /* send the message $TIME<CR> on a byte by byte basis */

     UPECHO(DOLLAR);
     UPECHO('T');
     UPECHO('I');
     UPECHO('M');
     UPECHO('E');
     UPECHO(CR);


} /* end of timerr */
```

```
/*
 *
 *
 * MSGREPLY - Routine to write out caracters to <CAB> &
 *            <OPT>.
 *
 *          Routines called : UPECHO
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : July 12, 1986
 *
 */


#include SBCDEF.H


MSGREPLY(msg)    /* write characters to CAB & OPT */
char *msg;
{
   int i;
   int c;  /* convert data character to int */


      for (i = 0; msg[i] != EOS; i++) {

            c = msg[i];

            UPECHO(c);

      }

} /* end of MSGREPLY */
```

```
/*
 *
 * ALLECHO - Routine to write out a byte to <CAB>,<OPT>,
 * <STR>, & <PWR>.
 *
 *        Routines called : UPECHO, DOWNECHO
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : July 12, 1986
 *
 */


/* note - use outport(port, byte) */


ALLECHO(byte)       /* write a byte to CAB, STR, PWR & OPT */
int byte;
{


     upecho(byte);

     downecho(byte);


} /* end of ALLECHO */
```

```
/*
 *
 * UPECHO - Routine to write out a byte to <CAB> & <OPT>.
 *
 *        Routines called : OUTCAB, OUTOPT
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : July 21, 1986
 *
 */

/* note - use outport(port, byte) */


#include SBCDEF.H


UPECHO(byte)         /* write a byte to CAB & OPT */
int byte;
{


     outcab(byte);          /* outport(CAB, byte) */

     outopt(byte);          /* outport(OPT, byte) */


} /* end of UPECHO */
```

```
/*
 *
 * DOWNECHO - Routine to write out a byte to <STR> & <PWR>.
 *
 *          Routines called : OUTSTR, OUTPWR
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * Revision date : July 21, 1986
 *
 *
 */

/* note - use outport(port, byte) */


#include SBCDEF.H


DOWNECHO(byte) /* write a byte to STR & PWR */
int byte;
{


     outstr(byte);          /* outport(STR, byte) */

     outpwr(byte);          /* outport(PWR, byte) */


} /* end of downecho */
```

```
/*
 *
 * EVMMSG - Routine to send a fake message from environmental
 * module.
 *
 *        Routines called : UPECHO
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date : Dec. 11, 1986
 *
 */


#include SBCDEF.H

    extern int  cmd[];


EVMMSG() /* fake environmental module message */
{

     UPECHO(DOL);           /* send message to CABLE */
     UPECHO('B');           /* $ B B ... */
     UPECHO('B');
     UPECHO(cmd[0]);
     UPECHO(cmd[1]);

     if (cmd[1] == CR)        /* case 1 : {R,Z,T,L,B,O} */
          return;
     else {
          UPECHO(cmd[2]);
          UPECHO(cmd[3]);
          UPECHO(cmd[4]);
          UPECHO(cmd[5]);

          if (cmd[5] == CR) /* case 2 : F HEX(0) ... HEX(3)*/

               return;
```

```
            else {     /* case 3 : D HEX(0) ... HEX(3) 1 1 */

                 UPECHO('1');
                 UPECHO('1');
                 UPECHO(CR);

            }

       }

} /* end of EVMMSG */
```

```
/*
 *
 * ALLMSG - Routine to fake "all call" reply message.
 *
 *       Routines called : UPECHO
 *
 * author John F. Babson,
 * University of Hawaii, Physics
 * Revision date: Nov. 17, 1986
 *
 */


#include SBCDEF.H

    extern int cmd[];
    extern int dev;


ALLMSG() /* fake "all call" reply message */
{


      UPECHO(DOL);
      UPECHO(dev);
      UPECHO(dev);

      UPECHO(cmd[0]);
      UPECHO(cmd[1]);
      UPECHO(cmd[2]);
      UPECHO(cmd[3]);

      UPECHO(CR);


} /* end of ALLMSG */
```

```
/*
 * ASCBIN - Routine to convert a ASCII character to its
 * binary image.
 *
 *       Routines called :
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 * revised by John F. Babson
 * Revision date : July 9, 1986
 *
 */

#define NUMOFF 0X30
#define LETOFF 0X41


ascbin(c) /* converts a ASCII character to its binary image
*/
char c;
{
    int binary; /* value to be returned */

     /* check to see if char c is a number 0..9, if not then
        it is a letter A..F */

     if (c <= '9')
          binary = c - NUMOFF;      /* subtract the ascii
                                       offset */
     else
          binary = c - LETOFF + 0X0A; /* subtract the ascii
                                         offset and start
                                         count at 0A hex */

     return (binary);

} /* end of ascbin */


#undef NUMOFF
#undef LETOFF
```

```
/*
 *
 * BINHEX - Routine to convert byte sized binary numbers into
 * to their two byte ascii hex equivalent.
 *
 *       Routines called : NIBHEX
 *
 * author - John F. Babson,
 * University of Hawaii, Physics
 * revision date - July 20, 1986
 *
 */


#define LOMASK  OXOF /* low nibble mask */
#define UPMASK  OXF0 /* high nibble mask */


binhex (inbin,outhup,outhlo)

char inbin;     /* input binary byte */
char *outhup;   /* high ascii hex byte */
char *outhlo;   /* low ascii hex byte */

{

     char upnib;      /* high binary nibble */
     char lonib;      /* low binary nibble */


     /* divide inbin into the two nibbles upnib and lonib */

     lonib = inbin & LOMASK;

     upnib = inbin & UPMASK;

     /* shift right four bits of upnib to place the bits in
        lowest four bit position */

     upnib = upnib >> 4;
     upnib = upnib & 0x0F;
```

```
        /* finish the conversion passing the nibbles to nibhex*/

        *outhup = nibhex(upnib);

        *outhlo = nibhex(lonib);

} /* end of binhex */

#undef LOMASK
#undef UPMASK
```

```
/*
 *
 * NIBHEX - Routine to take a four bit binary nibble and
 * convert it into an ascii hex character.
 *
 *       Routines called :
 *
 * author - John F. Babson,
 * University of Hawaii, Physics
 * revision date - June 18, 1986
 *
 */


char nibhex (nibble)

char nibble;

{

     char value;


     if (nibble > 9) {

          value = nibble + 'A' - 10;

     } else {

          value = nibble + '0';

     }

     return (value);


} /* end of nibhex */
```

```
/*
 *
 * ISHEX - Routine to test a character if it's a ASCII hex
 * number.
 *      It returns TRUE if it's <0-F>, otherwise returns
 *      FALSE.
 *
 *      Routines called :
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii, Physics
 *
 */


#include SBCDEF.H


ISHEX(c)        /* returns TRUE if it's 0-F, otherwise returns
FALSE */
char c;
{

    if ((c >= '0' && c <= '9') || (c >='A' && c <= 'F'))

        return(TRUE);

    else

        return(FALSE);

} /* end of ISHEX */
```

```
/*
 *
 * INIT - Routine to initialize the SBC control program.
 *      It does the following :
 *          -   disables the HD64180 asychronous
 *              communications ports interrupts.
 *          -   sets the baud rate for all serial
 *              communication to 300 baud.
 *          - loads default parameters into the SBC latch.
 *
 *      Routines called :
 *
 * author - John Babson,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : Jan. 19, 1987
 *
 */


#include SBCMON.H

#define    ROM  1     /* 1 if TRUE (ROM system),
                         0 if FALSE (DOS system) */

   /* extern char table[]; */


INIT()
{
     /* disable the ASCI interrupt on the CAB port and set
        both the CAB and STR ports to 300 baud */

     #asm

            DB    0EDH,38H,STAT1  ; SB180 IN0 A,(STAT1)
            ANI   11110111B ; Receive Interrupt Enable - off
            DB    0EDH,39H,STAT1  ; SB180 OUT0 (STAT1),A

            MVI   A,00001101B     ; Baud Rate - 300
            DB    0EDH,39H,CNTLB0      ; SB180 OUT0 (CNTLB0),A
        IF    ROM   ; change to 300 baud only if ROM
        DB    0EDH,39H,CNTLB1 ; SB180 OUT0 (CNTLB1),A
        ENDIF

        ; note - default for console is 9600 baud

    #endasm


    /* now, set the SBC default parameter values */

    /* note - need to define the default values */

    /*
    table[3] = ....
    table[4] = ....
    table[5] = ....
    table[6] = ....
    table[7] = ....
    table[8] = ....
    table[9] = ....

    tablat();
    */

} /* end of init */
```

```
/*
 *
 * DEINIT - Routine to return to the SB180 Monitor.
 *         It does the following :
 *         - resets all error flags (OVRN, FE and PE) to
 *         zero.
 *         - enables the asynchronous communications ports
 *         interrupts.
 *         - force return to SB180 Monitor.
 *
 *        Routines called :
 *
 *        Notes:
 *         - value of label DOS determines proper consol baud
 *         rate upon return to DOS or ROM monitor
 *
 * authors Y. Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revision date : Sept. 17, 1986
 *
 */


#include SBCMON.H

#define DOS     0    /* 1 if TRUE, 0 if FALSE */

deinit()   /* enable ASCI interrupt - ch.1 */
           /* reset all error flags        */
{

    #asm

           DB    0EDH,38H,CNTLA1       ; SB180 IN0 A,(CNTLA1)
           ANI   11110111B  ; Error Flag Reset - off
           DB    0EDH,39H,CNTLA1       ; SB180 OUT0 (CNTLA1),A

           DB    0EDH,38H,STAT1  ; SB180 IN0 A,(STAT1)
           ORI   00001000B  ; Receive Interrupt Enable - on
           DB    0EDH,39H,STAT1  ; SB180 OUT0 (STAT1),A
```

```
        IF   DOS
        MVI  A,00001000B      ; Baud Rate - 9600 for DOS
        DB   0EDH,39H,CNTLB1        ; SB180 OUT0 (CNTLB1),A
        ENDIF

        RET                 ; Force return to SB180 Monitor

    #endasm

} /* end of deinit */
```

```
/*
 *
 * DELAY - Routine to cause one millisecond of time delay.
 *
 *      Routines called :
 *
 * authors John F. Babson and  Y. Miyakoshi,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : July 17, 1986
 *
 */


delay() /* delay 1 millisecond */
{

     #asm

          MVI  A,200      ; counter 200 times

     LOOP:
          NOP
          NOP
          NOP

          DCR  A          ; decrement counter
          JNZ  LOOP

     #endasm

} /* end of delay */
```

```
/*
 *
 * ATOD - Device driver routine for the ADC0816 16 channel
 * a to d converter chip.
 *
 * author - John F. Babson, University of Hawaii, Physics
 * revision date - July 23, 1986
 *
 */


#include SBCMON.H


int atod (chan)

{

     chan; /* pass the channel identity to be read to ASM
             level */

     #asm

          ; set up stack for parameter passing

          POP  D     ; return address

          POP  H     ; channel identity (chan)

     START:

          MOV  A,L   ; select channel address

          ; write out the channel to atod chip starting
          ; conversion process

          DB   0EDH,39H,ADC1   ; SB180 OUT0 (ADC1),A
```

```
    WAIT:

         ; read status - is end of conversion true ?

         DB    0EDH,38H,ASTATUS   ; SB180 IN0 A,(ASTATUS)

         ANI   EOC   ; check for EOC condition

         ; DANGER !!! - should add a timeout to prevent a
         ; system hangup

         JZ    WAIT  ; busy wait until EOC condition set

    READ:

         ; read in the byte

         DB    0EDH,38H,ADC1   ; SB180 IN0 A,(ADC1)

         MOV   L,A   ; store "byte" (in <L>)

         MVI   H,0   ; pad <H> with all zeros

         ; restore the stack

         PUSH H     ; return value (c)

         PUSH D     ; return address

    #endasm

} /* end of atod */

/*
 *  note: calling sequence -
 *
 *   chan = ascbin(channel)
 *   ;  [chan is binary, channel is ascii hex]
 *
 *   value = atod(chan);  [value is binary value returned]
 */
```

```
/*
 * incab - routine to input a byte from the optical cable
 * (CAB)
 *         note, much of this code is modified from the SB180
 *         monitor
 *
 * authors - John Babson and David Harris,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi - May 28, 1986
 * revision date - Nov. 1986
 *
 */

/*
 * assembly note: options for stripping parity and checking
 * for control character input exist and can be enabled or
 * disabled accordingly by commenting out (or not) the
 * appropriate lines marked by a "==>" pointer
 * in the inline comment area.
 */

#include SBCMON.H


int incab() {

     #asm

     ;note - ASCI interrupt's been disabled in main driver

          ;set up stack for parameter passing

          POP  D          ; return address
          POP  H          ; return value (c)

          MVI  B,6        ; initialize timer counter
                          ;   5 msec of time delay
```

```
INCAB1:
     DB    0EDH,38H,STAT1  ; SB180 IN0 A,(STAT1) -
                           ; check status
     ANI   80H         ; receiver not ready
     JZ    DELAY       ; null data

     DB    0EDH,38H,RDR1   ; SB180 IN0 A,(RDR1) -
                           ; read data byte
     ANI   7FH         ; ==> strip parity
     JMP   INCAB2      ; ==> return data byte directly
                       ;   skip SCHECK

SCHECK:
     CPI   18H         ; cntl-X
     JZ    RESTART         ; force a restart
     CPI   03          ; cntl-C
     JZ    RESTART         ; force a restart
     CPI   10H         ; cntl-P
     JNZ   INCAB2          ; default path
     PUSH  PSW         ; else
     LDA   ECHOFLG         ; toggle
     CMA               ; printer
     STA   ECHOFLG         ; flag
     POP   PSW
     JMP   INCAB2

RESTART:
     LHLD  ABORTMSG    ; reply with warm boot
     CALL  MPRINTF     ;   message
     JP    MONUP       ; and execute warm boot

ABORTMSG:
     DB    'WB00',0DH,0    ; warm boot message, CR

DELAY:
     DCR   B           ; select next timer counter
     MOV   A,B         ; move timer counter --
     JZ    NOCDATA     ; time out with null data

     MVI   A,200       ; loop counter for 1 msec delay
```

```
    LOOP:
         NOP
         NOP
         NOP
         DCR  A           ; decrement loop counter
         JNZ  LOOP        ; repeat until time out

         JMP  INCAB1      ; try again for input

    NOCDATA:
         XRA  A           ; clear <A>

    INCAB2:
         MOV  L,A         ; store "byte" (in <L>)
         MVI  H,0         ; pad <H> with all zeros

         ;restore the stack

         PUSH H           ; return value (c)
         PUSH D           ; return address

    ;note - ASCI interrupt will be enabled in main driver



    #endasm

} /* end of incab */
```

```
/*
 * instr - routine to input a byte from the string (STR).
 *      note, much of this code is modified from the SB180
 *      monitor
 *
 * authors - John Babson and David Harris,
 * University of Hawaii, Physics
 * revised by Y. Miyakoshi
 * revision date - Nov. 1986
 *
 */

/*
 * assembly note: options for stripping parity and checking
 * for control character input exist and can be enabled or
 * disabled accordingly by commenting out (or not) the
 * appropriate lines marked by a "==>" pointer
 * in the inline comment area.
 */

#include SBCMON.H


int instr() {

     #asm

     ;note - ASCI interrupt's been disabled in main driver

          ;set up stack for parameter passing

          POP   D          ; return address
          POP   H          ; return value (c)

          MVI   B,6        ; initialize timer counter
                           ;   5 msec of time delay
INSTR1:
     DB    0EDH,38H,STAT0  ; SB180 IN0 A,(STAT0) -
                           ; check status
     ANI   80H         ; receiver not ready
     JZ    DELAY       ; null data

     DB    0EDH,38H,RDR0   ; SB180 IN0 A,(RDR0) -
                           ; read data byte
     ANI   7FH         ; => strip parity
     JMP   INSTR2      ; => return data byte directly
                       ;   skip SCHECK
DELAY:
     DCR   B           ; select next timer counter
     MOV   A,B         ; move timer counter --
     JZ    NOSDATA     ; time out with null data
     MVI   A,200       ; loop counter for 1 msec delay
LOOP:
     NOP
     NOP
     NOP
     DCR   A           ; decrement loop counter
     JNZ   LOOP        ; repeat until time out
     JMP   INSTR1      ; try again for input

NOSDATA:
     XRA   A           ; clear <A>

INSTR2:
     MOV   L,A         ; store "byte" (in <L>)
     MVI   H,0         ; pad <H> with all zeros

     ;restore the stack

     PUSH H            ; return value (c)
     PUSH D            ; return address

;note - ASCI interrupt will be enabled in main driver

#endasm

} /* end of instr */
```

```
/*
 * inpwr - routine to input a byte from the power module
 * (PWR), one shot read version.
 *
 * authors - John Babson, David Harris, and Yoshiko Miyakoshi
 * University of Hawaii Physics
 * revision date - Nov. 11, 1986
 *
 */

/* assembly note: option for stripping parity exists and can
 * be enabled or disabled accordingly by commenting out (or
 * not) the appropriate lines marked by a "==>" pointer
 * in the inline comment area.
 */


#include SBCMON.H


int inpwr() {

    #asm
        ; set up stack for parameter passing

        POP  D          ; return address
        POP  H          ; return value(c)

        MVI  B,6        ; initialize timer counter
                        ;   5 msec of time delay

    INPWR1:
        DB   0EDH,38H,PSTAT  ; SB180 IN0 A,(PSTAT) -
                             ; check status
        ANI  80H        ; receiver not ready
        JZ   DELAY      ; null data

        DB   0EDH,38H,PDATA  ; SB180 IN0 A,(PDATA) -
                             ;read data byte
        ANI  7FH        ; ==> strip parity
        JMP  INPWR2     ; return data byte directly
    DELAY:
         DCR   B            ; select next timer counter
         MOV   A,B          ; move timer counter --
         JZ    NOPDATA      ; time out with null data

         MVI   A,200        ; loop counter for 1 msec delay
    LOOP:
         NOP
         NOP
         NOP
         DCR   A            ; decrement loop counter
         JNZ   LOOP         ; repeat until time out

         JMP   INPWR1            ; try again for input

    NOPDATA:
         XRA   A            ; clear <A>

    INPWR2:
         MOV   L,A          ; store "byte" (in <L>)
         MVI   H,0          ; pad <H> with all zeros

         ; restore the stack

         PUSH H             ; return value(c)
         PUSH D             ; return address


    #endasm

} /* end of inpwr */
```

```
/*
 *
 * OUTCAB - Routine to output a byte to the optical cable
 * port.
 *
 *       Routines called :
 *
 * authors - John Babson and David Harris,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : June 10, 1986
 *
 *
 */


#include SBCMON.H


outcab(byte) {


     #asm

          OUTCAB0:

               MOV  A,L   ; write the "byte"
               PUSH PSW   ; <A> to CAB

          OUTCAB1:

               ; check status
               DB    0EDH,38H,STAT1  ; SB180 IN0 A,(STAT1)

               ANI   02    ; if TDR not empty
               JZ    OUTCAB1    ; then wait
               POP   PSW   ; if TDR empty

               ; send the data
               DB    0EDH,39H,TDR1   ; SB180 OUT0 (TDR1),A
```

```
                PUSH PSW
                LDA  ECHOFLG    ; echo printer?
                ORA  A
                JNZ  ECHO ; yes - then echo
                POP  PSW  ; no - return directly
                JMP  OUTCAB2

          ECHO:
                LDA  PRNSTAT    ; to centronics port
                ORA  A
                JZ   ECHO ; wait printer free
                LDA  0    ; mark printer busy
                STA  PRNSTAT

                DI
                POP  PSW
                OUT  CENTDC     ; output data - clear strobe
                OUT  CENTDS     ; output data - set strobe
                OUT  CENTDC     ; output data - clear strobe
                EI

          OUTCAB2:

     #endasm

} /* end of outcab */
```

```
/*
 *
 * OUTSTR - Routine to output a byte to the string port.
 *
 *       Routines called :
 *
 * authors - John Babson and David Harris,
 * University of Hawaii, Physics
 * Revised by Y. Miyakoshi
 * Revision date : June 10, 1986
 *
 */


#include SBCMON.H

outstr(byte) {

     #asm

          OUTSTR0:

               MOV  A,L    ; write the "byte"
               PUSH PSW    ; <A> to STR

          OUTSTR1:

               ; check status
               DB    0EDH,38H,STAT0  ; SB180 IN0 A,(STAT0)

               ANI   02    ; if TDR not empty
               JZ    OUTSTR1    ; then wait
               POP   PSW   ; if TDR empty

               ; send the data
               DB    0EDH,39H,TDR0   ; SB180 OUT0 (TDR0),A

     #endasm

} /* end of outstr */
```

```
/*
 * OUTPWR - Routine to output a byte to the power module
 * port.
 *
 *       Routines called :
 *
 * authors - John Babson, Dave Harris,and Yoshiko Miyakoshi
 * University of Hawaii, Physics
 * Revision date : July 20, 1986
 */

#include SBCMON.H

outpwr(byte) {

     #asm

          OUTPWR0:

               MOV  A,L   ; write the "byte"
               PUSH PSW   ; <A> to PWR

          OUTPWR1:

               ; check status

               DB   0EDH,38H,PSTAT  ;SB180 IN0 A,(PSTAT)

               ANI  40H   ; if XHR not empty
               JZ   OUTPWR1 ; then wait
               POP  PSW   ; if XHR empty

               ; send the data

               DB   0EDH,39H,PDATA  ;SB180 OUT0 (PDATA),A

     #endasm

} /* end of outpwr */
```

```
/*
 *
 * OUTOPT - Routine to output a byte on the 23 bit wide
 * serial data optical channel.
 *
 *       Routines called : DATLAT
 *
 * authors John F. Babson and Y. Miyakoshi,
 * University of Hawaii
 * Revised by Y. Miyakoshi
 * Revision date : July 18, 1986
 *
 */

    extern char table[];


OUTOPT(c) /* writes data on OPT of latch */
{

     char byte;


     byte = c;  /* convert integer to character */

     table[0] = byte;      /* load the byte */
     table[1] = byte;      /* load the byte */

     table[2] = table[2] & 0x80;     /* mask MSB */
     table[2] = 0xAA;      /* load ?0101010 */


     /* move the pattern into the latch & strobe the SBC */

     DATLAT();


} /* end of outopt */
```

```
/*
 *
 * SBCDEF.H - Header file of defines for the String Bottom
 * Controller program.
 *
 *      This must be included (as #include SBCDEF.H) in all C
 *      source code files in which any of these defines are
 *      used.
 *
 * authors John F. Babson and  Yoshiko Miyakoshi,
 * University of Hawaii, Physics
 * Revision date: Nov. 19, 1986
 *
 */

/* device defines */

#define POWER    'A'     /* power module */
#define EVM      'B'     /* environmental module */
#define LATCH    'C'     /* latch */
#define OM1      '1'     /* optical module 1 */
#define OM2      '2'     /* optical module 2 */
#define OM3      '3'     /* optical module 3 */
#define OM4      '4'     /* optical module 4 */
#define OM5      '5'     /* optical module 5 */
#define OM6      '6'     /* optical module 6 */
#define OM7      '7'     /* optical module 7 */
#define CM1      '8'     /* calibration module 1 */
#define CM2      '9'     /* calibration module 2 */
#define ALLMOD   '0'     /* all modules */
#define ALLCM    'E'     /* all calibration modules */
#define ALLOM    'F'     /* all optical modules */

/* flag defines */

#define ONCAB    0x01    /* on inflg(0) - 0000 0001 */
#define ONSTR    0x02    /* on inflg(1) - 0000 0010 */
#define ONPWR    0x04    /* on inflg(2) - 0000 0100 */
#define OFFCAB   0xFE    /* off inflg(0)- 1111 1110 */
#define OFFSTR   0xFD    /* off inflg(1)- 1111 1101 */
#define OFFPWR   0xFB    /* off inflg(2)- 1111 1011 */
```

```
/* common port defines */

#define CAB    0         /* optical cable */
#define STR    1         /* string */
#define PWR    2         /* power */
#define OPT    3         /* data word on latch */

/* latch address defines */

#define LATCH0  0xF0     /* first latch byte */
#define LATCH3  0xF3     /* fourth latch byte */
#define DATAST  0xF9     /* the data strobe pulse */
#define MASTST  0xFB     /* master reset strobe pulse */

/* software time out - note that in INPOLL that INCAB, INSTR,
 * and INPWR are all 5 msec delays so that one poll pass for
 * all three ports is 15 msec long.  Thus, a 5 sec wait is
 * about 300 passes X 15 msec
 */

#define MAXCNT  300      /* max count for time out */

/* others */

#define DOL     '$'      /* dollar sign */
#define DOLLAR  '$'      /* dollar sign */
#define CR      '\r'     /* carriage return */
#define EOS     '\0'     /* end of string */
#define NULL    0
#define TRUE    1
#define FALSE   0


/* end of SBCDEF.H */
```

```
/*
 *
 * SBCGLOB.H - Header file of declaration of all global
 * variables.
 *
 *   This file must be included in 'MAIN' procedure of the
 *   String Bottom Controller program (as #include
 *   SBCGLOB.H). Any of these global variables must be
 *   declared as 'extern' in all files in which that global
 *   is used.
 *
 * authors Yoshiko Miyakoshi and John F. Babson,
 * University of Hawaii, Physics
 * Revised by John F. Babson
 * Revision date: Jan. 22, 1987
 *
 */


    int  sbcrun; /* control flag for normally running    */
                 /*   SBC program if TRUE, else run      */
                 /*   SB180 Monitor                      */

    int  inbuf[3]; /* buffer for a input character:      */
                   /*   inbuf[0] inbuf[1] inbuf[2]       */
                   /*   <CAB>    <STR>    <PWR>          */

    int  inflg;  /* bit flag for a input character:      */
                 /* bitflg(bit0) bitflg(bit1) bitflg(bit2)*/
                 /*   <CAB>        <STR>        <PWR>    */

    int  dev;    /* defined device name                  */
                 /* (defines are in 'SBCDEF.H' file)     */

    int  cmd[8]; /* command from console:                */
                 /*   <cmd0> <cmd1> ... <cmd7>           */

    int  bytnum; /* index on command line data character */
                 /* (0-11)                               */
```

```
int  poll;    /* switch for Environmental Module Poll:*/
              /*   if 'poll' is 'EVM' do 'EVMPOLL'    */
              /*   else do 'CMDPOLL'                  */

int  talk;    /* switch for STRING "ALL CALL"         */
              /*   if 'talk' is TRUE do 'ALLPOLL'     */
             /*   else do 'CMDPOLL'                   */

int  time;    /* switch for STRING module response   */
              /*   if 'time' is TRUE do 'TIMPOLL'    */
              /*   else do 'CMDPOLL'                 */

int  timout; /* flag for controlling software time  */
             /*   out when a valid (format wise)    */
             /*   STRING or POWER command is passed */
             /*   on so if no response is received  */
             /*   by time out period the system will */
             /*   not hang waiting for input from   */
             /*   the port                          */

int  count;  /* count to control software time out  */

char table[9]; /* internal table of bytes -          */
            /*   corresponding to ports on latch    */
            /*      table[0] ... table[8]            */
            /*      latch[0] ... latch[8]            */

char command[2];       /* for debugging */

/* end of SBCGLOB.H */
```

```
/*
 *
 * SBCMON.H - Header file of defines for connecting up the
 * modified SB180 monitor program (renamed SBCMON.Z80) with
 * the String Bottom Controller program.
 *
 *          This must be included (as #include SBCMON.H) in
 *          all C source code files in which any of these
 *          defines are used.
 *
 * authors John Babson and David Harris,
 * University of Hawaii, Physics
 * Revised by Yoshiko Miyakoshi
 * Revision date: Aug. 19, 1986
 *
 */


/* common defines */

#define STAT0   04H  /* I/O port 0 (STR) status */
#define STAT1   05H  /* I/O port 1 (CAB) status */
#define MPRINTF 13C8H     /* Z80 monitor PRINTF */
#define PSTAT   0E1H /* I/O POWER status */

/* input defines */

#define RDR0    08H  /* receive data port 0 */
#define RDR1    09H  /* receive data port 1 */
#define ECHOFLG      0FF2DH     /* echo flag for printer */
#define MONJP   0369H     /* monitor jump (restart address) */
#define PDATA   0E0H /* receive data POWER */

/* output defines */

#define TDR0    06H  /* transmit data port 0 */
#define TDR1    07H  /* transmit data port 1 */
#define PRNSTAT 0FF2FH    /* printer status */
#define CENTDC  0C0H /* centronics "clear" */
#define CENTDS  0C1H /* centronics "set" */
```

```
/* other I/O addresess */

#define CNTLA0  00H   /* ASCI control register A0 (AUX) */
#define CNTLA1  01H   /* ASCI control register A1 (CON) */
#define CNTLB0  02H   /* ASCI control register B0 (AUX) */
#define CNTLB1  03H . /* ASCI control register B1 (CON) */

/* ATOD device driver defines */

#define ADC1     0E2H       /* atod enable line */
#define ASTATUS 0EFH        /* atod status line */
#define EOC      02H        /* atod EOC condition 00000010B */


/* end of SBCMON.H */
```

REFERENCES

PREFACE

Childress, White, and Walter, Fun With Our Family, Scott, Foresman, and Company, Glenview, Illinois, 1965 (good Dick and Jane example)

CHAPTER 1

Griffiths, David, Introduction to Elementary Particles, Harper and Row, Publishers, New York, 1987

CHAPTER 2

Alexander, G. et al., Phys. Lett. **78B**, 162 (1978)

Bacino, W., et al., Phys. Rev. Lett. **42**, 749 (1979)

Blocker, C.A., Dorfan, J.M., et al., Phys. Lett. **109B**, 119 (1982)

Feldman, G.J., Trilling, G.H., et al., Phys. Rev. Lett. **48**, 66 (1982)

Particle Properties Data Booklet, April 1988 from "Review of Particle Properties", Phys. Lett. B **204** (April 1988)

CHAPTER 3

Alexander, G. et al., Phys. Lett. **78B**, 162 (1978)

Bacino, W., et al., Phys. Rev. Lett. **42**, 749 (1979)

Blocker, C.A., Dorfan, J.M., et al., Phys. Lett. **109B**, 119 (1982)

Li, X. and Ma, E., Phys. Rev. Lett. **47**, 1788 (1981)

Particle Properties Data Booklet, April 1988 from "Review of Particle Properties", Phys. Lett. B **204** (April 1988)

Tsai, Y.S., Phys. Rev. D **4**, 2821 (1971)

Weinberg, Steven, The First Three Minutes, A Modern View of the Origin of the Universe, Basic Books, Inc. New York, New York , 1977

CHAPTER 4

Bosetti, P., et al., DUMAND II Proposal to Construct a Deep-Ocean Laboratory for the study of High Energy Neutrino Astrophysics and Particle Physics, UH Preprint #HDC-2-88, Hawaii DUMAND Center, University of Hawaii, July 27, 1988

Jackson, John David, Classical Electrodynamics, 2nd Edition, John Wiley and Sons, New York,1975

Matsuno, S., et al., Nuc. Inst. Meth., **A276**, 359 (1989)

CHAPTER 5

Alekseyev, E.N. et. al., Baksan Underground Scintillation Telescope, Proc. 16th ICRC, Kyoto, vol. 10, pp. 276-281 (1979)

Alexeyev, E.N., Alexeyeva, L.N., Chudakov, A.E., and Krivosheina, I.V., Status of the Baksan Experiment on the Search for Neutrino Bursts from Stellar Collapses, Proc. 20th ICRC, Moscow, pp. 277-280 (1987)

Andreyev, Yu.M., Gurentsov, V.I., and Kogai, I.M., Muon Intensity from the Baksan Underground Scintillation Telescope, Proc. 20th ICRC, Moscow, pp. 200-203 (1987)

Bionta, R.M. et al., Observation of a Neutrino Burst in Coincidence with Supernova SN1987a in the LAR... Phys.Rev.Lett.58, 1494 (1987)

Bionta, R.M., et. al., The IMB Search for Stellar Collapse, Proc. 20th ICRC, Moscow, vol. 6, p. 317, 1987

Chen, H.H., Kropp, W.R., Sobel, H.W., and Reines, F., Muons Produced by Atmospheric Neutrinos: Analysis*, Phys. Rev. D, 4, 99 (1971)

Chudakov, A.E. et. al., Study of High Energy Cosmic Ray Neutrinos, Status and Possibilities of Baksan Underground Scintillation Telescope, Proc. 16th ICRC, Kyoto, vol. 10, pp. 287-292 (1979)

Crouch, M.F., Landecker, P.B., Lathrop, J.F., Reines, F., Sandie, W.G., Sobel, H.W., Coxell, H., and Sellschop, J.P.F., Cosmic-ray muon fluxes deep underground: Intensity vs. depth, and the neutrino-induced component, Phy. Rev. D, **18** (1978)

Grant, A.L., Review of the Status of Proton Decay Experiments Outside the USA, Proceedings of the 1982 Summer Workshop on Proton Decay Experiments, June 7-11, 1982, Argonne National Laboratory, pp. 203-223, Ayres, D.S. (ed), 1982

Haines, T.J., et al., Calculation of Atmospheric Neutrino Induced Backgrounds in a Nucleon Decay Search, Phys. Rev. Lett. **57**, 1986-1989 (1986)

Hirata, K.S., et. al., Experimental Study of the Atmospheric Neutrino Flux, Phys. Lett. D, **205**, 416 (1988)

Ikeda, H., et. al., KEK - (UT)2 Experiment on Proton Decay, Proc. 17th ICRC, Paris, vol. 7, p. 185, 1981

Kielczewska, D., et al., Experimental Limits on the Nucleon Lifetime from the IMB Detector, In *Kazimierz 1985, Proceedings, Elementary Particle Physics*, 383-395

Krishnaswamy, M.R., Menon, M.G.K., Narasimham, V.S., Hinotani, K., Ito, N., Miyake, S., Osborne, J.L., Parsons, A.J. and Wolfendale, A.W. The Kolar Gold Fields neutrino experiment I. The interactions of cosmic ray neutrinos, Proc.Roy. Soc. Lond. A. **323**, 489 (1971)

Krishnaswamy, M.R., Menon, M.G.K., Narasimham, V.S.,Hinotani, K., Ito, N., Miyake, S., Osborne, J.L., Parsons, A.J. and Wolfendale, A.W. The Kolar Gold Fields neutrino experiment II. Atmospheric muons at a depth of 7000 hg cm-2 (Kolar), Proc. Roy. Soc. Lond. A. **323**, 511 (1971)

LoSecco, J.M., et. al., A Study of Atmospheric Neutrinos with the IMB Detector, Proc. 19th ICRC, La Jolla, vol. 8, pp. 116-119, 1985

Menon, M.G.K., Naranan, S., Narasimham, V.S., Hinotani, K., Ito, N., Miyake, S., Craig, R., Creed, D.R., Osborne, J.L., and Wolfendale, A.W. Studies of cosmic ray neutrino interactions in the Kolar Gold Field experiment, Proc. Roy. Soc. A. **301**, 137 (1967)

Menon, M.G.K., Naranan, S., Narasimham, V.S., Hinotani, K., Ito, N., Miyake, S., Craig, R., Creed, D.R., Osborne, J.L., and Wolfendale, A.W. The Kolar Gold Fields neutrino project, Canadian Journal of Physics, **46**, S344 (1968)

Oyama, Y., et. al., Experimental Study of Upward Going Muons in Kamiokande, ICR-178-88-24, 20 p., Oct. 1988

Reines, F., IV High Energy neutrinos underground: status of the Case-Wits-Irvine experiment and future prospect, Proc Roy Soc A **301**, 125 (1967)

Reines, F., Kropp, W.R., Gurr, H.S., Lathrop, J., Crouch, M.F., Sobel, H.W., Sellschop, J.P.F., and Meyer, B. Measurements of interactions of cosmic-ray neutrinos, Can. Jour. of Phy, **46**, S350 (1968)

Reines, F., Kropp, W.R., Sobel, H.W., Gurr, H.S., Lathrop, J., Crouch, M.F., Sellschop, J.P.F., and Meyer, B., Muons Produced By Atmospheric Neutrinos: Experiment*, Phys. Rev. D, 4, 80 (1971)

Svoboda, R., et. al., The IMB Proton Decay Detector, Proceedings of the NATO Advanced Study Institute on Composition and Origin of Cosmic Rays, Sicily, Italy, June 20-30, 1982, pp. 363-366, Shapiro, M.M. (ed.), 1983

CHAPTER 6

Jennings, Fred, Practical Data Communications - Modems, Networks and Protocols, Blackwell Scientific Publications, Oxford, U.K., 1986

Roden, Martin S., Digital Communication Systems Design, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1988

Taub, Herbert and Schilling, Donald L., Principles of Communications Systems, McGraw-Hill Book Company, New York, 1971

Signetics Analog Timer Manual, Signetics Corporation, Sunnyvale California, circa 1977

TMS99532 Application Report, Texas Instruments, July 1983

CHAPTER 7

THE COMPLETE MOTOROLA MICROCOMPUTER DATA LIBRARY, Motorola Corporation, Phoenix, AZ, 1978

LINEAR DATABOOK, National Semiconductor Corporation, Santa Clara, CA, 1982

MCS-51 Family of Single Chip Microcomputers User's Manual, Intel Corporation, Santa Clara, CA, July 1981

Microcontroller Handbook, Intel Corporation, Santa Clara, CA, 1983

MM54HC/74HC HIGH SPEED microCMOS LOGIC FAMILY DATABOOK, National Semiconductor Corporation, Santa Clara, CA, 1983

Photomultiplier Handbook, RCA Corporation, Lancaster, PA, 1980

CHAPTER 8

THE COMPLETE MOTOROLA MICROCOMPUTER DATA LIBRARY, Motorola Corporation, Phoenix, AZ, 1978

LINEAR DATABOOK, National Semiconductor Corporation, Santa Clara, CA, 1982

MCS-51 Family of Single Chip Microcomputers User's Manual, Intel Corporation, Santa Clara, CA, July 1981

MCS-51 MACRO ASSEMBLER USER'S GUIDE, Intel Corporation, Sanata Clara, CA, 1983

Microcontroller Handbook, Intel Corporation, Santa Clara, CA, 1983

MM54HC/74HC HIGH SPEED microCMOS LOGIC FAMILY DATABOOK, National Semiconductor Corporation, Santa Clara, CA, 1983

Photomultiplier Handbook, RCA Corporation, Lancaster, PA, 1980

CHAPTER 9

Cummings, W. C., The Mythical Manmonth, Addison-Wesley, New York, 1974

IEEE, The 2nd Software Engineering IEEE Conference (Jack Tarr Hotel), San Francisco, California, 1976

Intel Semiconductor Corporation, The 8051 Users' Manual, Intel Inc., San Jose, California, 1982

Jensen, Kathleen and Wirth, Niklaus, PASCAL User Manual and Report, Springer-Verlag, New York, 1979

Kernighan, Brian W. and Plauger, P. J., Software Tools, Addison-Wesley Publishing Company, Menlo Park California, 1976

Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978

Leventhal, Lance A., Z80 Assembly Language Programming, Adam Osborne and Associates, Inc., Berkeley, California 94710, 1979

Lewis, T. G. Software Engineering for Micros, The Electrifying Streamlined Blueprint Speedcode Method, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1979

Motorola Semiconductor Products, M6800 Microprocessor Programming Manual, Motorola Inc., Phoenix, Arizona 85036, 1976

CHAPTER 10

Bilofsky, Walt, C/80 Small C Compiler, The Software Toolworks, Sherman Oaks CA, 1981

Cain, Ron, A Small C Compiler for the 8080's, Dr. Dobb's Journal of COMPUTER Calisthenics & Orthodontia, Running Light Without Overbyte, People's Computer Company, Menlo Park CA, No. 45, Vol.5, Issue 5., pp. 5-19, May 1980

Cain, Ron, A Runtime Library for the Small c Compiler, Dr. Dobb's Journal of COMPUTER Calisthenics & Orthodontia, Running Light Without Overbyte, People's Computer Company, Menlo Park CA, No. 48, Vol.5, Issue 8., pp. 4-15, September 1980

Ciarcia, Steven A., Build the SB180 Single-Board Computer, Part 1: The Hardware, Byte, McGraw-Hill, Hightstown, NJ, Vol. 10, No. 9, pp. 87-101.

Ciarcia, Steven A., Build the SB180 Single-Board Computer, Part 2: The Software, Byte, McGraw-Hill, Hightstown, NJ, September 1985, Vol. 10, No. 10, pp. 87-101, September 1985

Conn, Richard, ZCPR3 The Manual, New York Zoetrope, Inc., New York, 1985

Digital Research, CP/M 2.0 User's Guide, Digital Research, Pacific Grove CA, 1979

Hitachi America Ltd., HD64180 8-Bit High Integration CMOS Microprocessor User's Manual, Hitachi America Ltd., San Jose CA, October 1985

Kernighan, Brian W. and Plauger, P.J., Software Tools, Addison-Wesley Publishing Company, Reading Mass., 1976

Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978

Microsoft Corporation, Microsoft MACRO-80 ASSEMBLER Software Reference Manual, Microsoft Corporation., 1979

Richie, D.M., Johnson, S.C., Lesk, M.E., Kernighan, B.W., The C Programming Language, Dr. Dobb's Journal of COMPUTER Calisthenics & Orthodontia, Running Light Without Overbyte, People's Computer Company, Menlo Park CA, No. 45, Vol.5, Issue 5., pp. 20-29, May 1980

SLR Systems, SLRNK+ Superlinker Plus User's Guide, SLR Systems, Butler PA, 1985

The Micromint, Inc., SB180 Single Board Computer Users Manual, The Micromint, Inc., Vernon Conn., 1985

CHAPTER 11

Babson, J. et al., Cosmic Ray Muons in the Deep Ocean, UH Preprint #HDC-1-89, submitted to Phys. Rev. D (1989)

Crookes, J. N., and Rastin, B. C., Nuc. Phys. B, 58, 93 (1973)

Kobayakawa, K. private communication (1987)

APPENDIX C

Griffiths, David, Introduction to Elementary Particles, Harper and Row, Publishers, New York, 1987