DOE/NASA/0310-1
NASA CR-174723

DEVELOPMENT OF A MICROPROCESSOR CONTROLLER FOR STAND-ALONE PHOTOVOLTAIC POWER SYSTEMS

Alan R. Millner and David L. Kaufman
TriSolarCorp.
Bedford, Massachusetts 01730

(NASA-CR-174723) DEVELOPMENT OF A MICROPROCESSOR CONTROLLER FOR STAND-ALONE PHOTOVOLTAIC POWER SYSTEMS Final Report (TriSolar Corp.) 192 p HC A09/MF A01 CSCL 10B G3/44 N86-21979 Unclas 05822

June 1984

Prepared for

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, OH 44135

For

U.S. DEPARTMENT OF ENERGY
Conservation and Renewable Energy
Division of Photovoltaic Energy Technology
Washington, D.C. 20545
Under Interagency Agreement DE-A101-79ET20485

DOE/NASA/0310-1
NASA CR-714723

DEVELOPMENT OF A MICROPROCESSOR CONTROLLER FOR STAND-ALONE PHOTOVOLTAIC POWER SYSTEMS

Alan R. Millner and David L. Kaufman
TriSolarCorp.
Bedford, Massachusetts 01730

June 1984

Prepared for
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, OH 44135

For
U.S. DEPARTMENT OF ENERGY
Conservation and Renewable Energy
Division of Photovoltaic Energy Technology
Washington, D.C. 20545
Under Interagency Agreement DE-A101-79ET20485

FORWARD

The authors wish to acknowledge the excellent software development effort of Dr. Thomas Maiër and others of the staff of the Mellon Institute.

NASA FINAL REPORT

DEVELOPMENT OF A MICROPROCESSOR CONTROLLER FOR STAND-ALONE PHOTOVOLTAIC POWER SYSTEMS

TABLE OF CONTENTS

LIST OF FIGURES

1.0 EXECUTIVE SUMMARY

1.1 THE PROBLEM

Photovoltaic systems need controls of varying scope depending on the system size and application. While low cost voltage regulators are adequate and available for small battery chargers, many larger or more complex systems require control algorithms which until now could only be achieved by costly custom engineered products. The problem is made more difficult because the modular nature of photovoltaic systems results in a great variety of controller requirements.

As the cost of photovoltaic modules decreases, system control and related engineering costs become very important. To meet this need for flexibility with a minimum of customizing cost, and yet consume only a few watts of power is within the capabilities of today's microprocessor.

The first unit was produced under D.O.E. contract DEN3-310 and tested for NASA at TriSolarCorp. This unit is photographed in Figure 1.1-1. It is a 5kW maximum power controlling battery charger which can also be configured as a 5kW batteryless motor drive.

The entire controller interfaces with existing 500 watt MPC power modules, operates in a NEMA-4 enclosure over an ambient temperature range of -25 degree C to +45 degrees C, operates from either a 12V battery or a 40V to 300V DC unregulated solar array or battery bus.

1.2 HARDWARE DESCRIPTION

The prototype microprocessor PV system controller consists of a NEMA-4 enclosure containing the following major electronic assemblies:

A. Power Supply
B. Display and Control Interface
C. Resistor Divider Board
D. Power Modules (10)
E. Processor

The power supply consists of a wide input range DC to DC down converter and a multi-output flyback DC to DC converter. The downconverter accepts 40V to 300V DC.

The display board consists of a 4 1/2 digit LCD display, a 16 key membrane switch pad (calculator style), an audible alarm, and three LED's: RED, YELLOW, and GREEN. The keypad allows manual control (under password security) of the system and calls up any one of over 50 different quantities to be displayed. The display normally shows battery state of charge but can display the voltage, current, or power of any array string, load bus, battery, motor, or other monitored point, monitored temperatures, or time of day. The LED's summarize system status: The GREEN LED shows that the processor is running and servicing timer interrupts. The RED LED indicates that the battery is dangerously low and loads have been shed, and the YELLOW LED indicates that stored energy is in short supply and should be conserved.

The resistor divider board serves as an interface between the high voltages in the power system and the low voltage signals measured by the processor.

The power modules are 500 watt DC to DC converters used, for the prototype, to couple the 180 volt nominal solar array inputs to the 120 volt nominal battery and load bus. These are 20kHz switching type bucking converters, with typical efficiency of 98%. They include internal fast current limiting. Ten power modules are provided in the prototype.

The processor is a large board containing the following parts:

1. Central Processor: CPU, timer, ROM, RAM
2. Analog input section: differential current, multiplexer, single-ended voltage multiplexer, and 10 bit plus sign A/D converter
3. Display and control interface: UART for RS232 interface support; PIA for keypad, LED, LCD, and interrupt support; 6 optoisolators for load control
4. Array control buffers: 10 digital outputs which can be used as pulse width modulated 20kHz signals for the power modules for maximum power point tracking, or can be used as DC level digital switch drive signals for discrete array control.

The central processor is built around a 65C02 CPU, which operates at 1MHz with an 8 bit data bus and a 16 bit address bus. This addresses 16k bytes of CMOS EPROM and up to 6k bytes of CMOS RAM. (This program requires only 2k of RAM.) The heart of the analog input section is the AD7571 analog to digital converter. This new device is a CMOS low cost unit, with 10 bits plus sign accuracy, and internal interface to an 8 bit microprocessor bus.

The RS232 port is provided by an IM6402 CMOS UART, which with a few discrete driver elements, makes a complete bi-directional asynchronoeus communications interface. This port allows data logging to an external printer, automated testing during manufacture, or computer interfacing in special applications.

The array control functions consist of a set of latches, gates, and buffers allowing 12 digital outputs to be driven as independent relay drive signals for discrete array control, or as separately buffered pulse width modulated 20kHz signals for maximum power point tracking. A watchdog timer turns off the array control and load outputs if the processor fails to reset it within 240msec. An additional analog output, 0 to 10V, is available for continuous analog control of variable loads such as variable speed motor drives. This is realized by low pass filtering of a second PWM signal.

1.3 SOFTWARE DESCRIPTION

The software design was based on two decisions. First, the software is modular. This allows an orderly linking of software modules using the principles of structured programming and allows modules to be modified without interfering with the rest of the program. Second, the modules were defined using "pseudocode", an explicit definition of each algorithm in plain English with a program format. This allowed clear communication between system designer, hardware designer, and software designer in advance of actual programming and eased the task of writing assembly language code.

The MAC unit is designed to operate without needing any human intervention. It performs the following functions automatically:

1. Battery Control
2. Array Control
3. Load or auxiliary display on the control panel
4. Status display on the control panel
5. Self-test
6. Data logging via RS232C port

In addition, the MAC unit provides the following facilities for manual control:

1. Manual battery charge initialization
2. Manual array control
3. Manual load or auxiliary charger control
4. Control panel display of system operating parameters
5. Manual test calibration mode and time of day
6. Remote control and parameter measurement or automated testing via the RS232C port

Each function is described below, with the automatic and manual functions described together for ease of understanding.

1.3.1 Battery Control

The state of charge describes the battery's condition at an instant in time. The initial battery state of charge can be entered manually from either the keyboard or the RS232C terminal. This can be based on a hydrometer reading of battery acid density, or simply an estimate. If it is wrong, the system will eventually correct it without damage to the battery. Default value at power-up is 50%.

The state of charge is increased by the number of ampere-hours flowing into the battery, and decreased by the number of ampere-hours flowing out of the battery. Rate of increase is modified by a tabulated coulombic charging efficiency which depends on state of charge. If the battery voltage exceeds expected limits, the state of charge is automatically adjusted to account for the variation.

For every ampere-hour of discharge by the battery, one equalization fraction ampere-hour of extra or equalization charging is automatically programmed when energy becomes available and 100% state of charge has been reached. This is needed to bring all cells up to full charge and to stir the acid by generating small amounts of gas. This prolongs battery life.

If the battery goes through many partial charge and discharge cycles before equalizing, the total equalization charge is limited to the equalization total percentage to avoid excessive water useage.

During charging and equalization, the battery is allowed to rise to the equalization voltage. After equalization is completed, the battery voltage is limited to the float voltage, which is below the point of significant gassing and so uses up less water. This float level charge makes up for internal battery self-discharging.

1.3.2 Array Control

If the battery is not fully charged and equalized, the voltage limit is the equalization voltage. If fully equalized, the voltage limit is the float voltage. Until reaching voltage limit, the solar array charges the battery as much as the available energy allows.

There are two automatic modes of array control resident in the program. The first is discrete array control. This turns strings on, one per second as long as the battery voltage is less the 94% of its maximum limit. It makes no change until the battery voltage reaches 97% of its maximum limit. Between 97% and 100% of its maximum limit, strings are turned off one per second. Above 100% of maximum limit, all strings are turned off. The battery voltage used for this algorithm is averaged over 4 samples to minimize noise problems.

The second mode is maximum power tracking. The controller drives up to 10 power modules with a 20kHz pulse width modulated drive signal, 13 volts = high = OFF, 0 volts = ON. The pulse width is set in 1/4 microseconds increments. It is initialized to 0 and changed in small (1/4 microseconds) or large (2 microseconds) steps every 100 milliseconds. At each step, the total delivered power is calcualted and compared with the power delivered at the previous duty cycle. If the power changed by less than 1.5%, and if no limiting conditions were encountered, small steps are used, otherwise big steps are used to speed up the process.

If the power level was found to be constant or increasing, the next step is taken in the same direction as the last. If the power was found to be decreasing, the direction of step in duty cycle is reversed. Therefore, the duty cycle is adjusted to the maximum power operating point, and hunts there +/- one or two small steps, representing an operating point within +/- 1% of maximum power. This improved array control method is the subject of a patent application.

1.3.3 Load Control

The battery is assumed to be loaded by up to 5 separate load busses, with separate voltage sensing, current shunts in the negative load, and relay control responsive to the optoisolator output 1-5 of the controller. These may in fact be auxiliary chargers as well as loads.

Each load bus is also automatically monitored for excessive current, providing a built in circuit breaker function. Each load or charger bus can also be manually controlled from the keypad or the serial port.

It is often desirable to use more energy if it's going to be a sunny day. This may be to avoid short cycles of load useage or simply to increase energy useage efficiency. Therefore, between 8AM and 12Noon, if the sunlight level is high enough to provide at least 10% of the maximum charger current limit to the battery, all the battery corrected SOC levels for load shed and restoration are decreased by a constant preset SOC, called DELTA SOC. This allows loads to turn ON earlier on a sunny morning than would otherwise be the case.

The MAC can perform a control strategy to allocate available energy into two different forms of storage. One of these is electrical energy stored in a battery. The other is a form of "product storage." This might be water stored in a tank, or thermal storage of cold in a refrigerator, freezer or ice stored in an icemaker. To use this capability, it is necessary to be able to measure the amount of product stored. This combination of trip points creates a set of four states of the controller, defined by the amount of product stored and the battery state of charge. By setting various values for the boundary parameters, various priorities can be placed in the use of energy.

1.3.4 CONTROL PANEL

The control panel has a keypad for manual inputs, a 4-1/2 digit LCD display for selected quantities, a set of red, yellow, and green LED's for quick status summary, and an audio alarm. There is also an emergency switch located at the bottom of the unit which, when turned OFF, will turn OFF all array strings and loads. When turned ON, it will restart the program.

The LCD display can be thought of as a multimeter which will display one of a menu of system parameters. The default condition is state of charge, and after a data dump each half hour, it returns to this parameter setting. It can also display any array string voltage, any charger output string current, (not the array current, but the converted output from that string at the battery voltage level), the delivered string power to the battery, the battery voltage current or power, any of five load bus voltages or currents or powers, the battery temperature, one other temperature (used here for freezer product measurement), the pulse width, the equalization charge needed, the system zero voltage and 4.00 volt reference level, the temperature corrected state of charge, the time of day, and the software version number. These are selected using the keypad.

The keypad can also be used to change control functions. To do this, the correct password must be entered. Password accessible functions include setting a new load and restore thresholds for each load bus, setting initial state of charge and setting the manual control for each load bus ON or Off.

1.3.5 Self-testing

The autómatic tesing of battery SOC and load currects is described elsowhere. Also, the unit can be placed in a test/cal mode where data is refreshed but the states of charge, max power tracking and load controls do not automatically change. This allows them to be manually set and measurements taken. For example, a fixed pulse width can be set from the serial port, or a lamp test can be performed.

The display performs one more automatic test. If one of the charger power module output currents is less than a constant (1amp) below the average of all such currents, the "Continuity" segments of the LCD is turned ON. This means that one string of the array or one power module of the charger is either shadowed or broken. Examination of array voltages and currents will show the cause.

Besides these external tests, the unit performs an automatic test of its RAM and ROM access whenever it is started up. When running, any error causing the program not to execute properly will result in a watchdog timer fault turning off all loads and the solar array and causing the green LED to go out.

1.3.6 Serial Port Functions

The manual test/cal mode was partially described in the previous section. However, with an RS232C terminal, much more extensive manual testing is possible. A standard debug monitor is implemented which can examine memory locations, insert bytes into memory, begin execution at a given memory location, insert or remove breakpoints, and download and program into memory.

A data dump of all system voltages, currents, temperatures, switch positions and states of charge is sent to the serial port every half hour or whenever requested. A printer there will provide data logging.

This powerful access allows many possible uses. The processor can be automatically tested during manufacture this way. A printer attached at the site can serve as a data logger. A telephone modem or other communication interface would allow automatic remote data logging or even, remote system control.

The use of a small RS232C terminal by a repair technician allows fault isolation and diagnosis beyond the capabilities of the keypad and display, since small special test routines can be run. Software modification is greatly eased as well.

1.4 TEST RESULTS AND CONCLUSIONS

The battery charge control algorithm works as described. Its estimate of the fraction of capacity available is limited mainly by the accuracy of the battery's rated capacity at a given discharge rate which is supplied by the battery manufacturer. Long term low current operation is dominated by internal battery self-discharge, typically 1% to 2% per week, and current offset errors of 0.1% of maximum, which is often a similar percentage per week. This is corrected whenever the battery is fully charged, so that batteries cycled daily or weekly are not affected.

The array control modes, both max power tracking and discrete switching, work very well and provide a nice tapered current finishing charge for a battery with very low water loss. The max power tracking with two step sizes is a real advance in the state of the art, as it provides a combination of more accurate tracking of a static max power point plus a faster acquisition time for a varying load or insolation. A chart comparing these performance figures is given here.

COMPARISON OF MPC FUNCTIONS

	Previous State of the Art Equipment	MAC
Acquisition time +/-2%	3 sec	1 sec
Max Dither, stable load	3%	1%
Product	9	1

This means that, for example, a volumetric pump or other slowly cycling loads can be dynamically matched to the solar array with reduced need for expensive or lossly load-levelling mechanisms as large flywheels, batteries, etc. This opens up additional applications for the controller.

Load management in response to battery state of charge is very good. It is free of the chattering and instability characteristices of many voltage-related load management schemes. The algorithm for product storage and apportioning energy to several types of loads works as expected. However, its sophistication makes it difficult for a user to verify that it is operating properly, and this may make it less popular than other approaches based solely on battery status. Instrumentation of the system using the keypad and LCD display is very effective. A permanently posted list of command codes on the unit near the keypad was found to be useful.

Data logging via the RS232C terminal to an inexpensive printer is very helpful in village systems. Of less use in smaller applications, it is quite helpful for maintenance or fault diagnosis using a small hand held battery-powered RS232 terminal. The possibilities inherent in a phone coupler or other communicaions port for remote system control have not been explored, but they are potentially very interesting. The self-test functions of the controller are an effective means of allowing unskilled personnel to monitor a complex PV system.

In conclusion, the microprocessor automatic controller works as well as, or better than, any existing PV system control equipment, and can be adapted to a wide variety of systems simply by plugging in an EPROM. This makes it a very attractive controller for PV systems where large size (over 1kW), remote location or special control requirements justify this type of unit.

2.0 DESIGN CONSIDERATION

2.1 INDUSTRY NEEDS

The photovoltaic industry is growing today at an annual rate of nearly 65%. This growth includes a varied mix of small simple battery chargers, pump drives, large sophisticated battery chargers, and utility interactive systems. As the cost of the PV module decreased, the fraction of the system cost represented by controls and engineering becomes more important. In order to allow the continued growth of volume and reduction of system prices, a controller is needed which is flexible enough for all applications while easy to apply without high recurring engineering costs. The microprocessor represents an opportunity to supply that need.

In order to determine the range of system parameters required for a controller, a preliminary analysis of the near term PV market is useful. The following figure 2.1-1 indicates the breakdown of the 1983-84 PV market, using a variety of various documented and informal sources, to ascertain the relative market share of various types of systems. The first rows of figures give the breakdown by system size, first a typical size, then a range of sizes, by half-decade logarithmic steps. The percent of power sold represents the fraction of PV kilowatts used in that size system. The percent of systems sold gives the fraction of the total number of systems regardless of size. A typical installed system price in $/watt is indicated for reference.

The next section characterizes the controls, including power conditioning, for a PV system in terms of the dollar cost of controls at the OEM level divided the peak watts of PV in the system. These are divided into systems with Maximum Power Controllers (MPC) and those without. The difference represents the incremental cost of the MPC. The approximate fraction of systems without MPC's is indicated, and multiplied by the number of systems in each category, to give the number of controllers with and without MPC's in each system size category. This shows that the market for MPC type controller falls primarily in the 300 watt to 30 kW size range. The large number of systems at smaller sizes indicates the attractiveness of marketing a small, inexpensive controller which might be able to capture the 100 W to 300 W market even without maximum power control. The high cost of controls below 100 watts size indicates that any controller will be hard to sell for such small systems.

The last row of figures estimate the incremental value of an MPC for 1983 and 1986 PV systems. These estimates are based on the following rows of figures for typical system voltage, typical controllers and PV utilization, efficiency, and typical incremental PV array installed prices. Comparing these figures with the controller costs determines the utility of an MPC in such a system. Note that MPC's are probably worthwhile for systems over 300 watts at present and over 1 kW in 1986.

Based on these results, it appears that a controller intended for the

PRELIMINARY PV INDUSTRY 83-84 PARAMETERS
ESTIMATED 5 MEGAWATTS/YEAR, 10K SYSTEMS

Typ.(W)	50	200	500	2K	5K	20K	50K	200K
Range	100W	100-300	300-1K	1K-3K	3K-10K	10K-30K	30K-100K	100K
SYSTEMS								
% of power sold	5	10	15	15	20	15	10	10
% of systems sold	55	25	15	3	1.5	.38	.1	.02
Inst.$/watt cost	30	27	25	22	20	17	15	10
CONTROLS, $/W COST (INCLUDING POWER CONDITIONING)								
MPC	3.00	2.00	1.00	0.80	0.70	0.60	0.50	0.40
No MPC	2.00	1.00	0.60	0.40	0.30	0.20	0.15	0.10
% With MPC	0	1	5	15	20	40	50	50
# W/O MPC	5.5K	2.5K	1.5K	255	120	22	5	1
# W/ MPC	0	25	75	45	30	16	5	1
Incr.value of MPC 1983 $/W	-.30	.20	.70	1.0	1.20	1.20	.80	.80
1986 $/W	-.15	.10	.35	.50	.50	.60	.40	.40
Typ. Sys. Voltage	12	12-24	24-48	48-90	120	120	240	240
MPC effic.	.82	,87	.92	.95	.97	.97	.98	.98
No MPC effic.	.85	.85	.85	.85	.85	.85	.90	.90

Assume incremental PV array prices.

	$/W installed		
1983	1984	1985	1986
10	8	6	5

FIGURE 2.1-1

100 watt to 300 kW size range will market well if it:

1) includes MPC capability at an incremental cost of under $ .40/watt

2) is also useful for battery management without an MPC

3) costs less than $300 in a minimal configuration without the power electronics of an MPC. (The cost of a no-MPC controller for a 500W system)

The needs of the industry can then be segregated into at least two different types of controllers. The very small, very inexpensive controller (typically one quad comparator today) for systems of a few hundred watts or less is not the same as the flexible, sophisticated unit for larger systems. The small controller is discussed later as the "zero option". The larger controller can then be examined in terms of those characteristics which have a major affect on the unit cost.

2.2 MAJOR COST IMPACT ITEMS

2.2.1 System level items

The items affecting controller cost fall into two categories: those which describe the system, and those which describe the controller itself. These are listed in Figure 2.2-1. System characteristics affect the controller by the number of analog currents or voltages to be measured, by the number of counter/timer functions, and by the power or energy level of those parameters. This last group impacts the sizing of relays, power converters, shunts, and terminals in an obvious way which is separate from the control function itself; and is easy to design as required. In particular, any inputs or outputs of the control system which are continuously variable rather than switched are most expensive because they require either A/D or D/A converters, analog multiplexers, or counter/timer functions. Extra array measurements, maximum power controllers for the array, and variable loads (such as motor drives) are in these categories. Also, since battery control is the most complex function of the controller, dominating its accuracy and speed requirements, the number of batteries to be controlled independently is a major factor.

The effect of controller characteristics on cost requires further description of these tasks. Figure 2.2-2 lists major control functions.

2.2.2 Battery Control Considerations

The basic algorithm to be used for battery state of charge (SOC) estimation will be ampere-hour integration. Total ampere-hours into and out of the battery will be used to estimate change in SOC. Beyond this, a variety of checks, adjustments and corrections might have been used.

DESIGN CONSIDERATIONS HAVING STRONG EFFECTS ON THE DESIGN OBJECTIVES AND COSTS

I. PHOTOVOLTAIC SYSTEM CHARACTERISTICS

1. Photovoltaic array size
2. Type, number and total energy of storage elements
3. Type, number and total power of variable controlled loads

II. CONTROL SUBSYSTEM CHARACTERISTICS

1. Array control
2. Battery SOC algorithm
3. Load management
4. Product storage load management
5. Requirement for independent manual control override
6. Manufacturability
7. Adaptability
8. Maintainability skill, test equipment and MTTR
9. Reliability - MTBF

Figure 2.2-1

Major Functions of the Controller

1. Battery State of Charge estimation

   - amp hour accumulation
   - temperature compensated voltage limits and float voltage
   - MPC inhibit
   - automatic charge equalization

2. Maximum Power Controller

   - input filtering
   - comparison and duty cycle stepping
   - limits
   - direct digital PWM control
   - output buffering

3. Load Control

   - variable load setting
   - multiple fixed loads
   - response to SOC
   - backup generator control
   - product storage status response

4. User Service

   - keypad input scan
   - parameter settings
   - multimeter readout
   - SOC readout
   - diagnostic readout

5. Self Test Capability

   - Array and battery and load monitoring programs
   - Self monitoring programs

6. RS232C Interface

Figure 2.2-2

The coulombic efficiency of the battery is not 100%. As the battery becomes more fully charged, this value drops from near 100% until the battery is fully charged and equalized and the coulombic efficiency reaches zero. One correction then is to discount some fraction of charging current based on SOC. This was implemented.

The battery's self-discharge rate represents internal reduction of SOC without external current flow. This depends on battery chemistry, temperature, and age. Therefore, some estimate of internal self-discharge rate based on temperature might be used as an approximation. However, this was not included in the present unit because it is so dependent on manufacturing parameters of the battery.

Battery terminal voltage depends on age, temperature, SOC, rate and past history. However, at full charge and at full discharge, the I-V characteristic of the battery can be well established, with a suitable temperature correction. This means that at a temperature-corrected float voltage, the SOC is corrected to 100%, or at a minimum voltage the SOC is corrected to 0. Another alternative is to measure the current drawn at the float voltage at a given temperature and extend the charge cycle (lower the calculated SOC) if it is too high. This last was not implemented because it again is very battery dependent. The first alternatives were implemented, however.

Equalization of cells is done as a voltage-limited (temperature compensated) charge based on a number of ampere-hours required equal to a fraction F of ampere-hours discharged from the battery. After the completion of equalization, the charge may have been terminated to save water, or a lower voltage trickle charge may might be maintained. Based on our experience with systems, the latter was chosen.

An estimate of the time to next required battery watering of amount of water required could be made by the processor. This would be somewhat inaccurate, but might be of some use in warning of required maintenance before actual failure. This was not implemented.

The battery state of charge is defined to be the fraction of nameplate ampere-hour capacity which can be delivered at a nominal discharge rate before dropping below a voltage threshhold (typically 1.75 volts per cell). This changes with the battery temperature. (If the value is corrected for temperature, the result may exceed 100% on a warm day.) This value was chosen for load control functions, and for front panel display of system status.

Estimates of state of charge by combining voltage, current, and temperature using a ROM look-up table at intermediate SOC level has been found previously to be inaccurate during charging periods, and was not used. Another approach, modelling the battery internally by a series of coefficients representing internal circuit elements, has been proposed by researchers at Tel Aviv University. See for example "Measurement of the State of Battery Charge using Improved Loaded Voltmeter Test," E. Ofrey and S. Singer, IEEE Trans. in

Instrumentation and Measurement, Vol. IM-31 No.3, pp. 154-158, Sept. 1982. This approach was found to give good long term results, but only after a particular make and model of battery was characterized extensively. Since this implies a practical limit in the flexibility of the controller, this approach was not utilized either. Instead, a current integration (amp-hour meter) approach corrected for coulombic efficiency based on state of charge was used. This is in turn corrected if it exceeds the extreme temperature-compensated voltage limits expected for that battery; a much easier parameter to specify.

To measure battery current within half the rate of the internal self-discharge current, it was necessary to resolve to the nearest 550 hour rate. This compares with typical maximum charge/discharge rates of 4 hours typically, or 1 hour in an extreme case. Therefore resolution within 9 bits was required. Since the full scale range of the shunt may not correspond to the maximum rates (by a factor of 2 or 3), resolution to 10 bits minimum was required. 12 bit resolution would be nice but not absolutely necessary. If 100 millivolt shunts are used, this implies offsets near 25 microvolts will be visible. Therefore, the design should strive for 12 bit or 25 microvolt resolution of battery currents, with only 10 bits or 100 microvolts actually required. The use of a 10 bit plus sign A/D converter and 4X signal averaging was selected.

One major expansion of the controller's task would be to separately calculate SOC of multiple parallel battery strings. This takes up some extra RAM, processing time, and implies a higher level of system costs to implement for equalizong strings separately. However, reliability and cost savings warrant the added complexity in some cases. This was not implemented in order to keep controller cost down.

2.2.3 Array Control Considerations

The PV array must be controlled under the following circumstances:

1. If a battery is used and is fully charged, to avoid excessive water useage.
2. To avoid excess voltage on the load.
3. To allow the load(s) to be turned off if the array dconnects directly to it, as discussed under load control.
4. To allow a portion of the array to be safely repaired while the remainder of the system is running.
5. To avoid discharging the battery at night if no isolation diode is used.
6. To protect the array in case of fault.

Typical means of array control include:

1. Series contactors or switch elements, usually with series diodes in all but some low voltage configurations.
2. Parallel switch elements acroll the solar array, with a series diode batween array strings and the battery.

4. Switching DC-DC converter elements between array and battery usually providing maximum power control.

The control signals required by these schemes are different. Series or shunt regulators require a digital signal for each array string. A parallel shunt element might require no signals, or might have a digital input to modify the set point voltages for equalization. The switching converter requires a pulse width modulated square wave at the switching frequency, typically 2kHz to 100kHz being extreme values and 20kHz most common.

Instrumentation of such systems over and above what is needed for control, would normally provide one voltage and one current measurement for each array string. Since the instrumentation provisions are the same, design of the controller must concentrate on providing signals for control in the various cases. The first two cases (series or parallel) require the same signals. Since dissipative shunts are cost effective only in small systems, the third case has no impact on microprocessor system design. The last case is very well matched to microprocessor systems capabilities. Note that many systems are cost effective without built -in instrumentation, but with provision for multimeter readings via test points.

Therefore, the most useful way to deal with the series, shunt and switching converter array control options was to provide a different ROM sub-program for the same basic system design. The maximum number of independent subarrays to be controlled must be set and 5 optoisolated or 10 direct coupled output signals was chosen as good number. This number of optoisolated output signals must then be supplied, and could be used for either array inputs or load busses depending on system requirements and ROM programming.

Next, considering the option of maximum power control, the ability to distinguish small changes in the combined output voltages and current generally limits maximum power tracker performance. Therefore, careful treatment of the signal to noise ratio and the filtering of the input signals is important. This was carefully analyzed during design.

Typically, the power is an insensitive function of the duty cycle near a maximum. Therefore, a 2% change in duty cycle F produces a 1% change in output power. In the resistive load case, Vo and Io may each change 1/2%. Therefore, to find the max power point within 1%, 6 bit resolution of duty cycle F and 8 bit resolution of V and I are required. Since the system must run at a fraction of full scale, an extra 3 bits is needed. Therefore, use of a 10 bit plus sign A/D gives adequate resolution. Filtering must recognize the presence of 20kHz and its harmonics as switching noise, and must produce the cleanest differences possible at 100 msec intervals. This means averaging a number of samples in the 100 msec window. Four samples were averaged.

Timer capability for 8 bit duty cycle resolution requires, for 20 kHz carriers, a timer speed of 5 mHz. This is fast for CMOS, so perhaps a 7 bit capability would suffice, or a non-CMOS timer could be used. The non-CMOS timer was implemental.

It is necessary to recognize when some other limit is operating to disable the max power tracker. In such a case, no change in output occurs except noise, and the duty cycle should then move toward a nominal value or remain fixed. Remaining in a "dither" pattern was chosen.

Ability to manually move the duty cycle up or down can be useful for array string testing, although it is not often required in an operating system. Therefore a test/cal mode was implemented.

Based on our experience, it is felt that up to 10 power elements should be driven by one MPC controller, each with a separate buffered output. These may be assumed to have their outputs in parallel, with separate subarrays. Therefore, 10 buffered outputs, all with the same pulse width modulated signal were provided. 10 DC currents and one output voltage were sampled.

With present production power modules rated a 500W, TriSolarCorp could handle up to 5 kW with one controller. Larger systems could use multiple controllers or a larger power module (perhaps 3 to 5 kW) could be developed.

There are two basic cases in our experience when it is useful to have two max power controllers (MPC) operating from the same solar array. The first occurs with a shunt wound DC motor driving a pump or other device. The field winding requires power for start-up and so must have high priority on solar input from a portion of the PV array. After 30% sun or so, the field winding is fully excited and the rest of the power from this portion of the array can be used by the armature. The field usually represents 3% to 5% of the total power dissipation at full load. Use of separate arrays and MPC's results in waste of 6% to 10% of the system output at full run. Idealy, this can be recovered by using extra DC-DC converters to divert some power from the field portion of the array to the armature.

A second instance occurs when a water pump and battery charger are combined. The battery may be small and ideally, after the battery is charged, the water pump could be used for excess energy use. Separate arrays waste the PV output to the battery when it is fully charged. Running the pump off the battery requires a bigger battery and loses energy in the battery.

Ideally one set of DC-DC MPC drives would power the field of the motor or charge the battery until it reached a limit. Then another set of MPC drives would pull extra energy from the same PV strings for variable load use.

Each maximum power controller function added to the control unit requires whatever power conditioning elements are needed, extra output V and I sense lines, and extra PWM timer, an output on-off control bit, a few bytes of RAM, and a small anount of extra ROM, plus the program execution time on a 100 millisecond loop. Since the power conditioning can be used as needed, the inputs and timer are the biggest penalty to a minimal control element configuration. For the prototype, the second MPC was not implemented, but the provision of an extra timer was made to allow later implementation.

2.2.4 Load Control Considerations

A variable load would be controlled by a proportional analog signal of 0-10 volts. Therefore, one such output was provided, with programmable direction, and SOC programmable set points for load shedding and load restoration. Resolution is not important since only approximate energy useages need to be controlled.

Most loads are not continuously power controlled and up to 5 discrete outputs should be provided for. These would be 1 bit isolated outputs, with programmable SOC "ON" and SOC "OFF" set points. These could also include back-up generators or other such devices.

Manual override capability is probably best placed in the loads themselves rather than in software. However, for self-test purposes, artificial counting up and down of the SOC control signals should be possible. Therefore, a manual SOC setting capability was included.

At least one output circuit for latching relay drive should be provided. However, this was left for external hardware rather than internal software.

Multiple variable DC loads must be assumed to be in parallel, otherwise the number of PWM counters becomes unmanageable for a single CPU controller and controller networking is required.

AC loads, to be variable, would require a variable voltage and frequency inverter. Drive signals for the poles of such an inverter might be useful, but might be used for this function only when it is needed, for variable speed AC pumps or compressors. These were not included.

Product storage can occur in a number of forms. Examples of this are:

1. Water tankage (pumping).
2. Ice making (refrigeration).
3. Phase change thermal storage (refrigeration or heat pump).
4. Fertilizer making.
5. Water heating (resistive).
6. Salinity (water desalination).

Because of the diversity of types, the control interface must be relatively simple to be standardized. Typically, the product storage system must indicate when it is full and when it is empty. Provision for this can be flexibly accomodated with two relay contact inputs, one normally closed and one normally open. Either one in the abnormal state should cause the PV system to turn OFF, under the assumption that product storage capacity has been reached. In most cases, a pair of level switches or thermostats can provide the desired signals. The capability of accepting other analog inputs would be provided in hardware anyway. One algorithm for sharing power between an analog measured product and a battery was be developed.

Output to load control may have to operate over hundreds of feet in electrically noisy environments. Grounding of such signals can cause problems for the control system. Therefore, all discrete load control signals were optoisolated. In low cost applications, the optoisolators can be omitted to avoid excessive cost of unnecessary options. (This same consideration applies to discrete array control signals for series or shunt regulated systems. The same outputs can be used, with a programming change.)

Predictive load control algorithms must take into account primarily the state of charge of the battery, and secondly the time of day and present insolation. This allowed correction of the load shed threshold to a perhaps 10% lower value if the time were 8 to 11 AM and the array current was 20% of the maximum, for example. This would avoid innecessary cycling of loads. Further use of predictive control requires algorithm development and field experience which is a good subject for research, but was not currently available for implementation in this controller. The controller will facilitate the data logging required to obtain parameters for evolutionary development of these methods.

2.2.5 User Service Considerations

As a minimum, any controller must allow the user to turn the MPC ON and OFF, and display the battery status. Given the data available to this controller, it also provides readouts of all sampled voltages, currents, or temperatures. It also provides alarm functions for system malfunctions (for example: output over voltages, output over current,motor over temperature, array string output zero for more than 24 hours) and perhaps some diagnosis of problems (power module shorted). Most alarms are visual, with provision for an audio option for critical failures.

Also, parameters of the system might to some extent be field or user reprogrammable to allow a single stock controller design to be produced for distribution. These parameters might include state of charge, number of power modules or array strings, motor voltage, battery voltage or capacity. The use of a back-up battery for internal memory or electrically reprogrammable ROM (EPROM) becomes necessary for this. It was decided to provide manual reassignment of load priorities, but all other parameters were fixed in EPROM.

The interface itself waschosen to be a low cost ($5 to $10) membrane switch keypad for input, and a 4 1/2 digit display for output. A buzzer or loudspeaker would be optionally plugged in. These features were mounted on an inner hinged panel, behind the weatherproof NEMA front panel and in a location protested from user contact with any live parts.

This also allows initiation of self-diagnosis routines from the keypad, which could test all elements of the processor and interfaces.

Further diagnosis might be done via an RS-232 port. This port could also provides for data logging of the signals sampled by the processor. These included array string input voltages, output voltages, and MPC power module output currents. From our experience, these facilitate fault diagnosis and isolation as well as permitting delivered power and efficiency calculations.

2.2.6 Self Test Considerations

At the highest level, the processor tests the solar array strings to determine if one is weaker than the other. It warns if the battery is too deeply discharged. It announces if a load draws much current.

At the next level, the controller tests itself. A test of RON, RAM, and all output audible and visible alarms is performed on demand. A watchdog timer checks of the processor is running at all times.

To allow both production testing and field repairs, an RS-232 interface and a small monitor program is useful to exercise the data collection, processing, output control, and timing functions of the controller under manual control. With only the keypad input and LCD readout, a more limited manual checkout of the system and the controller is possible. This simply displays voltage and current on demand, and allows manual unit turn ON and OFF, plus a scan of all parameter settings in the controller.

A standard hexadecimal 16 keypad suffices for the development systems with two preliminary letter keystrokes for function and numbers 0-9 for string or parameter identification. In production quantities, the letters could be replaced by more explicit labels of the functions.

2.2.7 Other Considerations

In addition to the these functional requirements, the need for a product which is easily reproduced without custom drawings for each application is important. Also the adaptability of the unit to a particular system without large numbers of selector switches, ports, jumpers or custom trimmed resistors is vital. Extensive use of EPROM, for system parameter and algorithm specification was chosen as the most viable approach. The use of replaceable modules at the board level into the unit, was found to be necessary for user acceptance in locations where PV is to be used.

Finally, the unit has to be very reliable if it is not to compromise the inherent advantage of reliable electric power which makes PV systems attractive. This implies use of proper temperature range components and minimization of parts count.

2.3. OPTIONS DEFERRED

In designing the controller, a number of decisions were made which might be re-examined at some future date as component technology changes. These are described here to facilitate such future project planning.

2.3.1 "Zero Option"

In looking at the market analysis for PV controllers, it appears that there may be a place for a very inexpensive (well under $100) controller without many of the features needed in large systems. Such a controller would be simply a battery regulator, with a single array input and a single load output. It would be intended for systems from 100 watts to 1kW peak, where the site access cost for battery maintenance or replacement would make state of charge control worthwhile. It would have no instrumentation or data logging capabilities, no sequential load shedding, no maximum power point reacking, and would be designed only for 12 or 24 volt batteries.

However, it would provide a readout of battery state of charge based on ampere-hour accumulation and would provide better battery charge control than the ten dollar voltage limiter with temperature compensation which would compete with it. Such a controller would require very little memory (RAM or ROM) and could be implemented with an 80C48, 146805, or equivalent processor. Inputs for voltage, current and temperature of the battery would be adequate. Latching relay drives would be appropriate for input and output.

The economics of such a controller are marginal at present. In the future, sufficient volume (several thousand per year) would allow a masked-ROM memory and an inexpensive 8 bit multiplexed input, resulting in parts costs under $40. Until such time as this market becomes better defined, the viability of such a product is not clear. However, a specification has been generated for future reference. The concept is labelled the "Zero Option" because it is not one of the five recommended options for the next phase of this contract. Figure 2.3.1-1 gives these specifications.

2.3.2 Larger Power Stages for Maxpower Control

The TriSolarCorp MPC power modules available at present are rated 500W each. The control of systems larger than about 5kW requires that more than 10 such power modules be used. This can be accomplished using the present MAC microprocessor controller only if not all of them are monitored, or if more than one MAC be used, or if relays instead of MPC power modules are used. A better solution is to develop power modules of near 5kW power rating. This requires interfacing to the MAC with a pulse modulated waveform of the proper frequency, since 20K HZ implemented at present may be too fast for larger devices. Also the dynamics of the control functions must be compatible with the slower filters of a larger power module.

This development was beyond the scope of the present contract.

2.3.3 Multiple MPC Control

In many systems more than one max power control function is needed,

SPECIFICATIONS FOR "ZERO OPTION" CONTROLLER

Array Voltage:	+10.0V to +60.0V
Array Configuration:	Single input
Array Control:	Discrete ON/OFF
Single Ended Inputs:	Battery voltage, battery temperature sensed by 10k ohm thermistor
Differential Input:	$\pm$100mV battery current shunt
Control Inputs:	Relay contact closures, One NC, One NO
Battery:	12V or 24V Nominal, 10 to 10,000 ampere-hours, required for operation
Battery SOC Control:	Ampere-hour accumulator, accuracy $\pm$10%, precision $\pm$2%
Battery Voltage Limits:	Maximum absolute limit, maximum temperature-compensated limit, minimum absolute limit
Control Digital Outputs:	One for array, one for load, latching relay drives
Manual Entry or I/O:	None
Display:	Battery SOC, 2 1/2 digits, LCD

FIGURE 2.3.1-1

since more than one power flow path from an array to a load must be controlled. No extra hardware in the CPU would be needed to implement the feature, but extra power conditioning hardware would be needed as well as appropriate software. This was not implemented in the prototype since no specific use was foreseen at present.

2.3.4 Variable MPC Control

The ability to drive standard sumersible AC motors for pumping applications would enhance the controller. However, the relation between an AC motor and its variable speed drive is particularly intimate in terms of protective features and optimal drive strategies.

Implementation of this function was left for a separate unit, with a simple interface to the MAC controller via an analog control voltage 0-10V F.S. This task is well worthwhile when resources are available to support it.

2.3.5 Multiple Battery Strings

If a large system consists of several parallel-connected battery banks, it is possible to compute the state of charge of each one and check that each is sharing the load. However, this is not critical to the PV system control task and so was not implemented at this time, since it impacts the size and speed of the controller.

2.3.6 System Topologies

The most fundamental choice in applying a microprocessor controller to large PV systems is the overall topology. How many controllers will be used, and how will they communicate? Three basic topologies were stúdied. These are described as follows:

Distributed Controller

The distributed controller is conceptually a number of independent microprocessors, each performing a different function in the overall system and each located with the equipment it controls. For example, in a village power system some control elements might be array controllers or max power controllers for separate subarrays, another might perform battery management, another control load management functions, and another might perform data logging and display functions. Only a small amount of slowly changing information would be exchanged between them via a bus structure of some kind. Any one controller could function in a backup mode without any information from other controllers.

Star Controller

In the star configuration, a master controller operates the central elements of the system and communicates with remote slave units ar the ports of the "star". These slave elements are simpler, more sprcialized units and communicate only with the master controller. In a village power system, for example, the master unit might perform battery and load management and data logging functions while the array controls or max power controllers were slaves. Diagnostic displays might be divided among the elements as required. Information on battery status would flow from the master unit, and array status would return to it. Backup operation would require manual control of the battery and loads.

Central Controls

In a central control scheme, all control functions reside in one controller with no processor or "smart" controls in any other place. In a large village system, all array switches or max power trackers and loads have their operating states set by the central controller. All data logging and most of the instrumentation would also be centralized. This configuration allows the most complex algorithms since a;; data is available for all functions. It also provides the least redundancy and backup capability, unless the central controller has it built in.

Assuming a microprocessor of even minimal complexity at each mode, a trade-off study showed that the central topology was the least expensive for all but the smallest systems. This was basically because the cost of memory, power supply, enclosures, and so on was too high for very small units of the distributed or star networks. Therefore, the central topology was pursued as the basic requirement for the MAC controller.

However, it should be noted that the MAC can be used in either distributed or star configuration under some circumstances.

If a large system required independent operation of several parts for higher reliability, several MAC units could be operated independently. No information would be exchanged between them, unless further development of a multi-station RS232C bus were completed.

Also, if a system had a number of "dumb" controllers such as the low cost TriSolarCorp BCR unit, each of these with its subarray could be treated as a discreet-switched string and controlled and monitored by a sequentially switched segmented array control, a very good way to handle large low voltage DC systems. The MAC would provide everything except max power tracking for such a system. The three basic topologies are diagrammed in Figure 2.3.6-1.

As part of the trade-off of system topologies, a Failure Mode and Effects Analysis (FMEA) was done for each option. Figure 2.3.6-2 shows the system level failure modes of the unit and their results. Figure 2.3.6-3 takes these failure modes and compares the impact of each of failure on overall system operation. As is easily seen, the impact of distributed system failures is far less than star system

15 kW System Alternatives

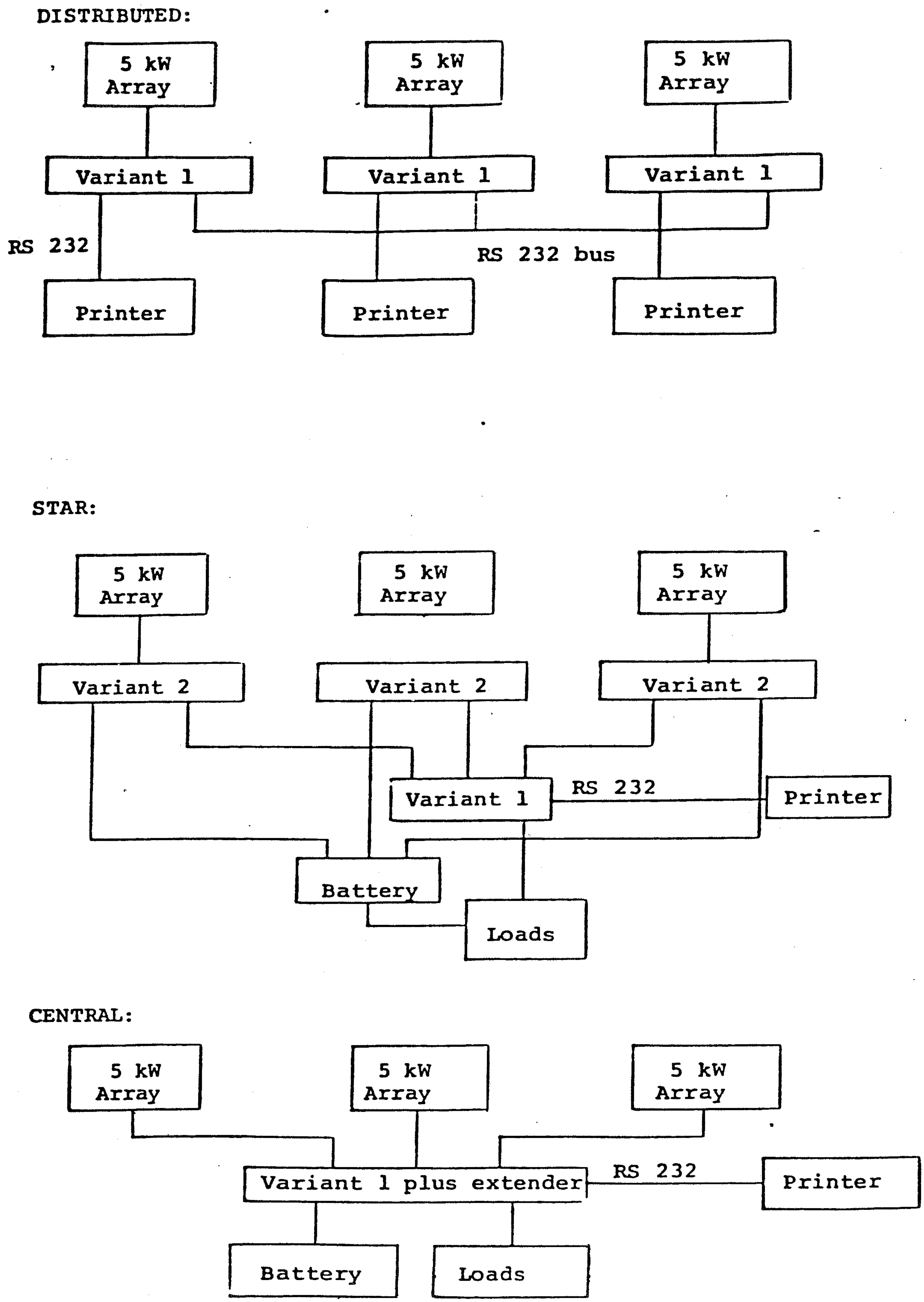


FIGURE 2.3.6-1

FMEA SYSTEM LEVEL

Function Lost	Distributed	Star	Centralized
Array Control			
Partial	Reduced output until scheduled maintenance	Reduced output until scheduled maintenance	Reduced output until scheduled maintenance
Total	Greatly reduced output until scheduled maintenance	Greatly reduced output until scheduled maintenance	Total shutdown controller replacement
Battery Control	manual operation modular replace-	manual operation master unit replacement	totally manual operation, controller replacement
Load Control	Manual load operation, modular replacement	Manual operation, Master unit replacement	Totally manual operation, controller replacement
Instrumentation	Operational, scheduled maintenance	Operational, scheduled maintenance	Manual operation Controller replacement
Data Logging & Communications	Manual operation, modular replacement	Manual operation, modular replacement	Operational, scheduled maintenance, modular replacement
Self Test	Operational, scheduled maintenance	Operational, scheduled maintenance	Operational, scheduled maintenance, controller replacement

FIGURE 2.3.6-2

FMEA IMPACT CHART

ANY ONE ELEMENT: FAILURE	DISTRIBUTED	STAR	CENTRAL
Power Supply			
Low	1	2	3
High	1	2	3
A/D Converter			
Calibration	1	1	2
No output	2	2	3
Display and Interface			
Partial	1	2	2
Total	2	3	3
Array Control			
Partial	1	1	1
Total	2	2	3
UART/RS-232			
Total	2	2	1
CPU & Memory			
Total	1 or 2	2	3
Current Meas. Max or Analog Mix			
Partial	1	1	2
Total	1	2	3

NOTE: 1=degraded performance, scheduled maintenance
2=manual operation until maintenance
3=total failure

FIGURE 2.3.6-3

failures, which in turn are less than central system failures.

2.3.7 Other Processors

The choice of processor is very fundamental to the realization of the controller. Based on the previous decisions, we needed a processor with 8-bit data bus, 16-bit address bus, and instruction times below 5 microseconds. A list of these are attached in Figure 2.3.7-1. In addition, memory requirements were estimated to be 2K of RAM and 12K of EPROM.

This was too much for the chips with on-chip memories which were available for the "zero option". Also, using a chip with all address and data lines available allowed system expansion for larger applications.

Major considerations then became power level (CMOS preferred), availability, support, and cost. These reduced the field to two major competitors, the national NSC 800 and the various suppliers of the 65C02. The main attractions of the NSC800 were availability and low power. In particular, the modular MA2000 version of the NSC800 looked attractive as a compact package. However, availability of the MA2000 in the time frame required was a problem, and the cost of the NSC800 in any form was higher because of the interfacing needed for its address and data busses. Development support for the 65C02 was more available at the facilities which needed it, and the cost was very low, the CPU only $10 in large quantities. After evaluations of samples from both GTE and Rockwell, this bacame the choice for the project. The availability of NMOS versions of most of the family at lower cost for breadboarding was also useful.

Analysis of the cost of the 65C02-based processor, which is suitable for any of the three geometries and can meet all the preliminary design requirements, shows that its hardware cost, in small quantity production, in materials alone will exceed the $300 cost target, meant to include labor and overhead as well. This problem motivated a careful look at lower cost alternatives which could be used in some topologies (perhaps as a peripheral unit of a star configuration) or with reduced capabilities. The number of analog inputs and load bus controls, plus the UART support requirements, were the driving force behind the processor choice. Therefore, a lower cost alternative was generated which approached the operational requirements of the system with less input and output.

The dramatic cost reduction possibilities lay in the use of a single chip computer with inboard EPROM. The chip selected for closest approach to the 65C02 in instruction set and largest memory capacity was the MC1468795G2. This has 2106 bytes of EPROM, 112 bytes of RAM, internal timer, 32 I/O lines, bootstrap programming and is plug compatible with a masked ROM version for future cost reduction in larger quantities. It combines CMOS power levels with a versatile instruction set. Its limitations for our application are expected to be the number of I/O lines and the amount of ROM. Its advantage is a cost reduction of almost a factor of two.

COMMERCIALLY AVAILABLE CMOS MICROPROCESORS WITH 8 BIT WIDE DATA PATHS AND INSTRUCTION TIMES BELOW 5 MICROSECONDS

ANUFACTURER	DEVICE	1. MEMORY (B) 2. RAM (C) 3. ROM (D)	INSTR CYC MIN/MAX (E)	INSTR #	INTERRUPT LEVELS	REGISTERS	MEMORY MAPPED I/O	ADDITIONAL FEATURES & COMMEN
OMMODORE	MCS-65C0X MCS-65C1X	64K	0.5/3.5	56	1	ACCUM. X, Y STACK PTR	YES	1. 6502 NMOS COMPATIBLE(MCS-6 2. HIGH LEVEL LANGUAGE SUPPOR EXTENSIVE
ITACHI	HD6301	64K 128 BYTES/ 2K BYTES	2/12	82	1	ACCUM A, B X, STACK PTR	YES	1. 6801 NMOS COMPATIBLE 2. MULTIPLY INSTRUCTION 3. 31 I/O LINES 4. DOUBLE PRECISON OP CODES
ITEL	MD68SC02AC	64K 128 BYTES	2/5	72	1	ACCUM A, B X, STACK PTR	YES	1. MOTOROLA 6802 NMOS COMPATI 2. 1V. STANDBY CAPABILITY 3-7V OPERATION
OTOROLA ITACHI	MC146805E2 MC146805G2 HD6305	8K 112 BTYES	2/4	61	1 IRQ TIMER SWI	ACCUM, X STACK PTR	YES	1. 32 I/O LINES 2. TIMER 3. 20MW ACTIVE, 1MW STANDBY 4. 3-6V OPERATION
C	NSC 800	64K	1.6/9.2	158	5	14GP	YES	1. Z 80 INSTRUCTION SET & CODE COMPATABILE 2. SUPERSET OF 8080/8085 SET 3. HIGH LEVEL LANGUAGE SUPPOR EXTENSIVE
C I C SHIBA	INS80C48 IN80CX48 MPD80C48 TMP80C48P	4K 64 BYTES 1K BYTES	2.5/5	36 37	1	16GP ACCUM	NO	1. SUBROUTINE NESTING LIMITED 8 LEVELS 2. 27 I/O LINES 3. TIMER 4. 8048 NMOS COMPATIBLE 5. NO HIGH LEVEL LANGUAGE SU
TEL C I SHIBA ERSIL	80C49 MPD80C49 TMP80C49P-6	4K 128 BYTES 2K BYTES	1.4/2.8	80	1	16GP	NO	1. SUBROUTINE NESTING LIMITE 8 LEVELS 2. 27 I/O LINES 3. TIMER 4. NO HIGH LEVEL LANGUAGE SU

COMMERCIALLY AVAILABLE CMOS MICROPROCESORS WITH 8 BIT WIDE DATA PATHS AND INSTRUCTION TIMES BELOW 5 MICROSECONDS (CON'T)

TOSHIBA NSC OKI	TMP80C3GP-6	4K 64 BYTES	2.5/5	96	1	16GP ACCUM	NO	1. 8039 NMOS COMPATIBI 2. 27 I/O LINES 3. NO HIGH LEVEL LANGU SUPPORT
RCA	1805	64K 64 BYTES	4/6	113	1	16GP	NO	1. 8 BIT COUNTER-TIMEF 2. SOME HIGH LEVEL LAN SUPPORT
INTERSIL	IM6100	4K	2.5/5.5	81	1	ACCUM MQ	NO	1. PDP-8 INSTRUCTION S 2. 4V TO 11V OPERATION 3. 12 BIT DATA PATH

NOTES: A) All units have expansion I/O capability
B) Memory direct addressing capability
C) RAM on CHIP
D) ROM on CHIP
E) In microseconds

FIGURE 2.3.7-1 B

To allow use of this chip, the array string currents are summed prior to sampling. The individual array string currents and voltages, if needed for instrumentation, must be switched manually to a metering input of the processor. Also, the two (not five) output bus lines are not measured automatically, although these too could be manually metered the same way. Only one PWM output is supported, with no analog output signal. Finally, the limited I/O of the chip will not support a UART for RS-232 linkage in addition to its other tasks. Communication via two dedicated digital inputs and one analog input is provided. Self test functions are minimized.

However, with these reductions, the processor can perform the key array control, battery charge control, load management, instrumentation, and self test functions in the system design requirements. The unit would be able to meet the requirements of the many small applications at a competitive cost, with the sacrifice of more complex system capability.

To meet more complex system requirements, the small processor option could be used in the following ways:

1. As a peripheral unit of a star configuration, with the two digital and one analog input lines to the hub element, the unit works. The central unit might be the 65C02 based controller or another small controller configured with no analog inputs but UART interface for data logging and a sinplifies link to the peripheral units.

2. As a multiple processor central unit, with one major function for each chip.

As a distributed processor element, the limited I/O capability makes the small processor option unfeasible. The small processor was not implemented in the prototype.

2.4 CONTROLLER SPECIFICATIONS

Having selected the important functions of the MAC unit, and deferred the options not to be included now, the requirements for the unit can be summarized as a specification. This is done in Figure 2.4-1.

Array Voltage	+10.0V to +18.0V or +40 to +300.0V, Input power under 5 watts
Array Configuration	1 to 10 string 40 to 500W per string
Array Control	Discrete string ON/OFF or maximum power control by Pulse Width Modulator (PWM) down converter
Tracking Accuracy	Within 1% of maximum output power

Control PWM Interface	12.5V (+1V/-.05V) pulse width modulated 20kHz CMOS logic level, up to 10 buffered outputs which can be connected to either one of 2 independent PWM drivers or are usable as 10 separate discrete control signals.
Single Ended Sense Inputs	+/- 4V DC full scale, +/- 2% accuracy, 10k ohm source impedance maximum. 16 voltage channels plus 2 thermistor channels
Differential Inputs	+/- 100mV full scale, +/- 2% accuracy, +/- 2V common mode maximum, 1K ohm source impedance maximum, 16 channels
Control Inputs	Two contact actuated inputs, one for normally open and for normally closed contacts. One manual system is shutdown. Normally open is ON. Contact rating required is 15V DC, 2 mA DC.
Battery	No battery is required, or with battery, 12V to 240V nominal, 100 to 10,000 ampere-hours.
Battery SOC Controls	Ampere-hour accumulator accuracy +/- 5%, precision +/- 1%.
Battery Voltage Limits	Maximum absolute limit, maximum temperature compensated charging limit, maximum temperature compensated float limit, minimum absolute limit, all programmable
Control Digital Outputs	Optoisolator outputs, 6 total; pairs may be used as latching relay drivers.
Control Analog Output	1 output +10V full scale, accuracy +/- 2% resolution 8 bits minimum.
Test and Logger Output	RS232 interface, 300 baud, full duplex Other rates programmable by EPROM change.
Display	LCD 4 1/2 digit display of battery SOC, any analog sensed input, calculated parameters, or error codes.
Manual Input	16-key keypad to select displayed quantities or self-test
Power Dissipation	5 watts or less for control elements only not including power modules or relays

Power to Other Elements	+12.5V (+1V/-0.5V) at 0 to 50 mA
System Loads	Battery, resistive DC, permanent magnet or wound field DC motor, inverter/battery
Ambient Temperature	Operating -25 degrees C to +45 degrees C, shipping -45 degrees C to +85 degrees C
Enclosure	NEMA-4

3.0 DESCRIPTION

3.1 HARDWARE DESCRIPTION

The prototype microprocessor PV system controller consists of a NEMA-4 enclosure containing the following major electronic assemblies:

A. Power Supply
B. Display and Control Interface
C. Resistor Divider Board
D. Power Modules (10)
E. Processor

A block diagram is shown in Figure 3.1-1. A photograph of the unit with internal cover open is shown in Figure 3.1-2. Schematic diagrams are attached in Appendix I.

3.1.1 Power Supply

The power supply consists of a wide input range DC to DC down converter and a multi-output flyback DC to DC converter. Total tare loss is 0.35 watts. The downconverter accepts 40V to 300V DC and produces 13.2 volts (referred to as a nominally +12V level) at up to 0.6 amperes, with an efficiency of approximately 64 percent. This can charge a 12 volt battery and can directly drive the flyback stage whether a battery is present or not. The flyback converter produces regulated +5V DC at up to 0.7 amperes, plus unregulated outputs of nominally -5V and -12V DC. Its efficiency at the +5V output is approximately 84% The processor logic runs on the basic +5V output, the analog signal conditioning amplifiers and RS232 interface run on +/-12V, the analog multiplexers run on +5V, and the power modules run on +12V. Total dissipation in the controller from a high voltage PV array, with NMOS EPROM's (CMOS units were not yet readily available) was 4.7 watts, distributed as follows:

Processor and display	2.0W
Power supply	2.3W
Power modules	10 at 40mW = 0.4W

The largest single power use is in the 9513 timer chip on the processor board, which uses about one watt at five volts. If a separate 12 volt supply or battery is used, thus avoiding down converter loss, total dissipation is only three watts.

The power supply is fully short circuit protected, all outputs are over-voltage clamped, and the five volt output has an over-voltage crowbar reset by power off.

3.1.2 Display and Control Interface

The display board consists of a 4-1/2 digit LCD display, a 16 key membrane switch pad (calculator style), an audible alarm, and three LED's: RED, YELLOW, and GREEN. The keypad allows manual control

PHOTOVOLTAIC SYSTEM BLOCK DIAGRAM

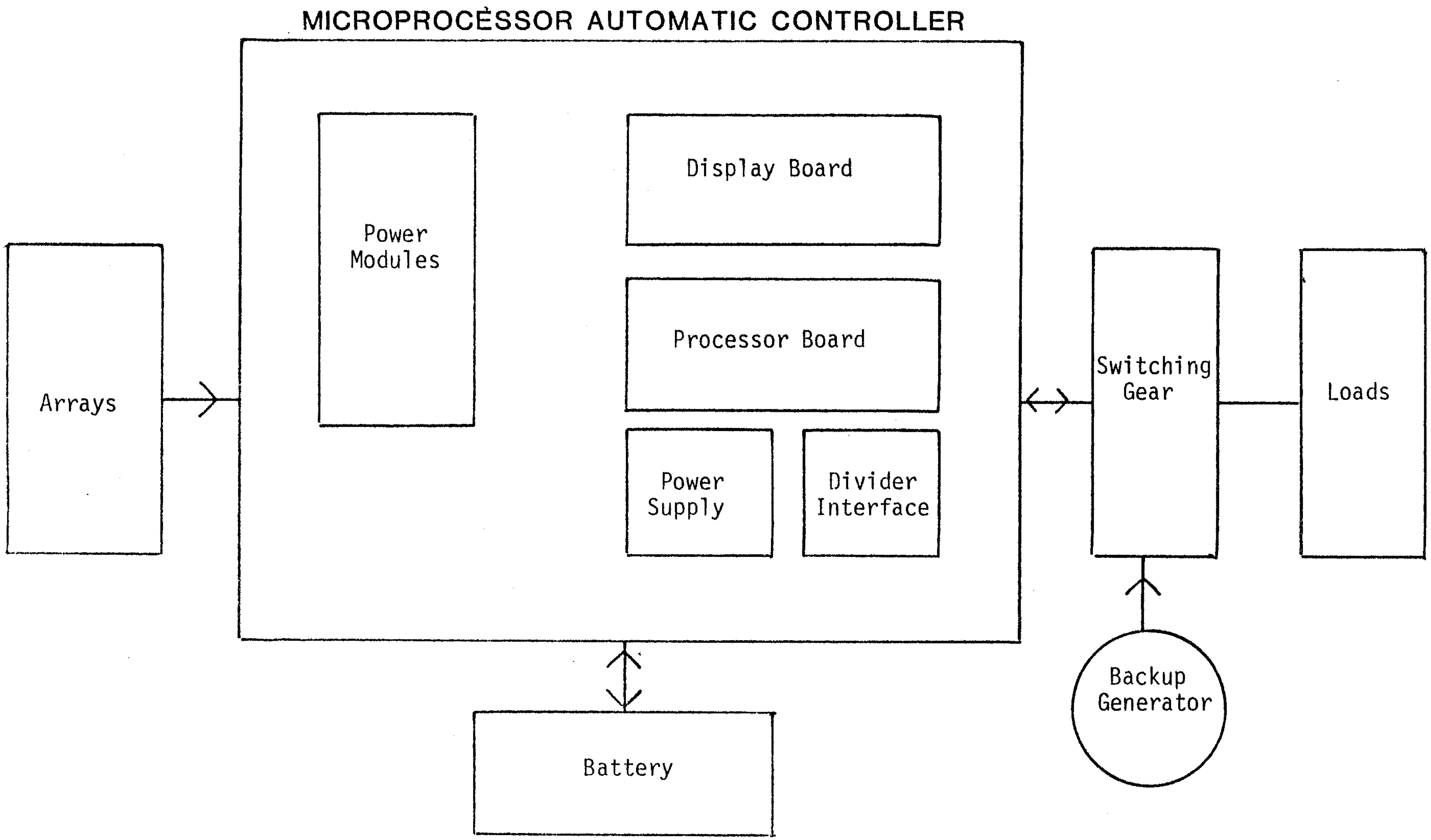


Figure 3.1-1

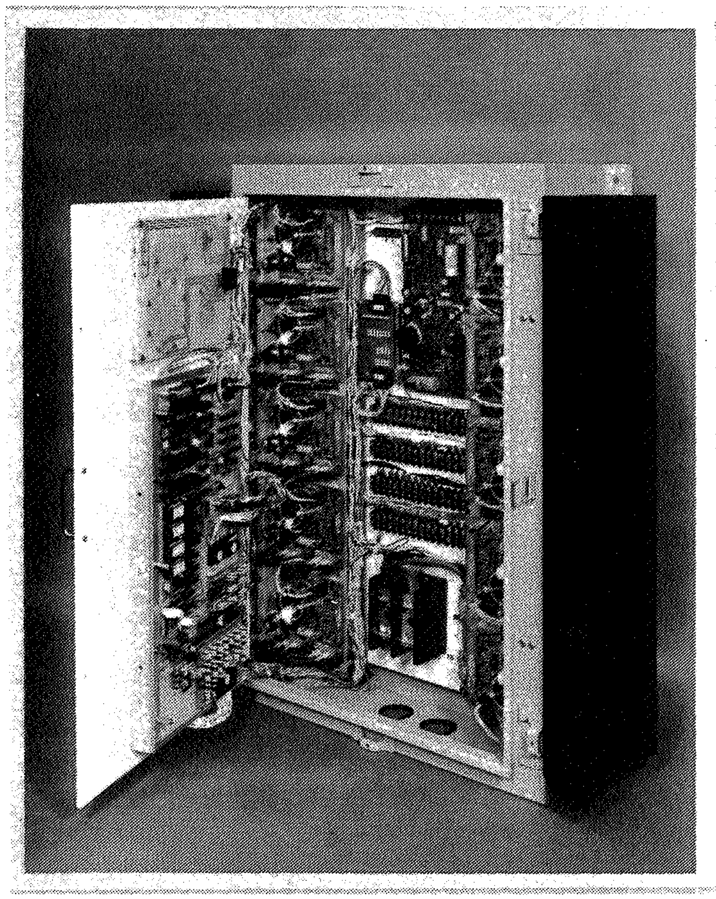


FIGURE 3.1-2
MICROPROCESSOR CONTROLLER

(under password security) of the system and calls up any one of over 50 different quantities to be displayed. The display normally shows battery state of charge but can display the voltage, current, or power of any array string, load bus, battery, motor, or other monitored point, monitored temperatures, or time of day. The LED's summarize system status: The GREEN LED shows that the processor is running and servicing timer interrupts. The RED LED indicates that the battery is dangerously low and loads have been shed, and the YELLOW LED indicates that stored energy is in short supply and should be conserved. A listing of the keypad-accessible functions is given in the software section of this report.

3.1.3 Resistor Divider Board

The resistor divider board serves as an interface between the high voltages in the power system and the low voltage signals measured by the processor. High voltages above 40 V are attenuated by a ratio of 100 to 1 before measurement while voltages from four volt to 40 volt full scale are attenuated by 10 to 1.

3.1.4 Power Modules

The power modules are 500 watt DC to DC converters used to couple the 180 volt nominal solar array inputs to the 120 volt nominal battery and load bus. These are 20kHz switching type bucking converters, with typical efficiency of 98 % They include internal fast current limiting. Ten power modules are provided in the prototype.

3.1.5 Processor

The processor is a large board containing the following parts:

1. Central Processor: CPU, timer, ROM, RAM
2. Analog input section: differential current, multiplexer, single-ended voltage multiplexer, and 10 bit plus sign A/D converter
3. Display and control interface: UART for RS232 interface support; PIA for keypad, LED, LCD, and interrupt support; six optoisolators for load control
4. Array control buffers: 10 digital outputs which can be used as pulse width modulated 20kHz signals for the power modules for maximum power point tracking, or can be switch drive signals for discrete array control.

The central processor is built around a 65C02 CPU which operates at 1MHz with an eight bit data bus and a 16 bit address bus. This addresses 16k bytes of CMOS EPROM and up to 6k bytes of CMOS RAM. (This program requires only 2K of RAM.) Address and data bus buffers separate the central portion of the processor from the rest of the controller. A multiple timer, the AM9513, is used to generate the 4MHz master crystal oscillator clock, 1MHz system clock, the time of day, the two pulse width modulated 20kHz outputs, the 16X baud rate clock for the RS232 port, and a 4msec interrupt used in the program. All "glue" chips (assorted small gates) are of the 74HC family for adequate speed at CMOS power levels. An address decoder for other major sections of the controller is also included in this section.

The heart of the analog input section is the AD7571 analog to digital converter. This new device is a CMOS low cost unit, with 10 bits plus sign accuracy, and internal interface to an 8 bit microprocessor bus. It is used in its "RAM mode", externally clocked at 500kHz, to give an 88 microsecond conversion time, with an external TL431 precision reference. Simple external address gating and buffering included here also provide multiplexing onto the data bus of converter status, as well as two time of day alarm outputs and two external control switch status bits.

The input to the converter comes from an analog multiplexer. This consists of a low offset buffer amplifier, channel selection latch, three 8-input single-ended CMOS multiplexers, and input filters. This services inputs for 16 voltage sampling channels, 16 currents, two thermistor inputs, a 4v reference, a ground reference, and the current measurement differential multiplexer. Each voltage sense input, with +/-4volt range, includes a diode clamped anti-aliasing filter to avoid averaging errors when sampling at the 4msec interrupt rate, and which also filters out RFI and protects against high voltage spikes. Filtering throughout the multiplexer system is extensive, but settling time is limited mainly by the slew rate requirements of the buffer amplifiers. Measured room temperature accuracy through the A/D converter is +/-0.2 percent.

Current sense inputs, with +/-100mV range, utilize a differential muliplexer with gain of 40. This consists of an address decoder, an instrumentation-amplifier style buffer amplifier, four 4-input dual CMOS multiplexers, and a differential anti-aliasing filter on each of 16 channels. This is also clamped for spike suppression and can tolerate +/-3volts common mode without significant accuracy degradation. This is needed to allow current shunts at reasonably remote locations in a large system, where wiring resistance drops can cause common mode offsets. Accuracy of 2% can be trimmed to better than 1%, with a DC common mode rejection ratio of 60db minimum.

Crosstalk in each stage of multiplexing is less than 0.1%, channel to channel, occuring with 10k signal sources at the voltage sense lines. This is limited by the "off" resistance of the low cost multiplexers used, and is adequate for this application but could be improved, since a worst case effective offset of just under +/-2% could result.

Anti-aliasing filter design of the current sense inputs is based upon the possibility of 100 or 120Hz square waves on the battery current line, sampled at the 4msec rate and averaged 16 times, resulting in an accuracy of better than five percent.

The display and control interface consists of a 65SC21 peripheral interface adapter (PIA) and an octal latch. The latch drives six 4N32 optoisolator outputs via a hex buffer and series resistors.

These resistors determine the output drive capability, and are presently set for 30 ma output drive.

The PIA provides two programmable 8 bit input/output ports and two pairs of dual programmable interrupt latches. The input/output ports service the LCD display, the keyboard, and the signals for the LED's and audio alarm, and are inputs for the UART status flags. The interrupts provide for the 4msec timer and the panic shutdown switch functions as well as UART service.

The RS232 port is provided by an IM6402 CMOS UART, which with a few discrete driver elements, makes a complete bi-directional asynchronoeus communications interface. At present, the parity, frame, and overflow error flags are ignored because a single system setting of baud rate and framing bits make these errors unlikely and error recovery in an unattended, stand alone system is impractical. This port allows data logging to an external printer, automated testing during manufacture, or computer interfacing in special applications.

The array control functions consist of a set of latches, gates, and buffers allowing 12 digital outputs to be driven as independent relay drive signals for discrete array control, or as separately buffered pulse width modulated 20kHz signals for maximum power point tracking. A watchdog timer turns off the array control and load outputs if the processor fails to reset it within 80msec. An additional analog output, 0 to 10 V, is available for continuous analog control of variable loads such as variable speed motor drives. This is realized by low pass filtering of a second PWM signal. Alternatively, the 12 array control outputs can be partitioned between the two PWM signals to provide max power tracking into two independent loads, although the present software doesn't yet fully support that mode of operation.

3.2 SOFTWARE DESCRIPTION

The software design was based on two decisions. First, the software is modular. This allows an orderly linking of software moudles using the principles of structured programming and allows modules to be modified without interfering with the rest of the program. Second, the modules were defined using "pseudocode", an explicit definition of each algorithm in plain English with a program format. This allowed clear communication between system designer, hardware designer, and software designer in advance of actual programming and eased the task of uniting assembly language code.

The MAC unit is designed to operate without needing any human intervention. It performs the following functions automatically:

1. Battery Control
2. Array Control
3. Load or auxiliary display on the control panel
4. Status display on the control panel

5. Self-test
6. Data logging via RS232C port

In addition, the MAC unit provides the following facilities for manual control:

1. Manual battery charge initialization
2. Manual array control
3. Manual load or auxiliary charger control
4. Control panel display of system operating parameters
5. Manual test calibration mode and time of day
6. Remote control and parameter measurement or automated testing via the RS232C port

Each function is described below, with the automatic and manual functions described together for ease of understanding. A complete pseudocode listing by subprograms is given in Appendix II. The key parameters for a sample system personalization of the MAC are given in Figure 3.2-1. A detailed listing of the object code is available in the Task III Interim Report.

3.2.1 Battery Control

The state of charge describes the battery's condition at an instant in time. The state of charge of the battery is defined to be a percentage equal to 100 times the number of ampere-hours the battery can deliver at 25 C at its specified rate, divided by the battery's rated ampere-hour capacity at 25 degrees centigrade. This is always less than 100 percent.

The corrected state of charge is defined to be a percentage equal to 100 times the number of ampere-hour capacity at 25 C. Note that this can be greater than 100 percent.

The initial battery state of charge can be entered manually from either the keyboard or the RS232C terminal. This can be based on a hydrometer reading of battery acid density, or simply an estimate. If it is wrong, the system will eventually correct it without damage to the battery. Default value at power-up is 50 percent.

The state of charge is increased by the number of ampere-hours flowing into the battery, and decreased by the number of ampere-hours flowing out of the battery. Rate of increase is modified by a tabulated coulombic charging efficiency which depends on state of charge as follows:

CONTROL PARAMETER LIST

MAC-P10 S.N. 001

	MIN	TYP	MAX	UNITS
Array Voltage	156	180.0	350.0	volts
Array String Current	---	2.0	2.5	amps
Number of Strings	---	10.0	-----	-----
Battery number of Cells series	54	54.0	54.	-----
Ampere-hour Capacity	---	400.0	-----	A-Hr
Float Voltage/Cell	---	2.4	-----	volts
Equalization Volts/Cell	---	2.7	-----	volts
Min. Voltage/Cell	---	1.9	-----	volts
Battery Current	-60	-----	+60	amps
Charger Current Limit	---	-----	51.2	amps
Load Current Limit	---	-----	60.	amps
Temperature Measurements	-5	25.0	60.	deg.C
Charger String Current Limit	---	6.5	-----	amps
Number of Load Busses	---	5.0	-----	-----

FIGURE 3.2-1

CHARGING EFFICIENCY

SOC	EFFICIENCY
0 TO .7	100
.7 TO .8	.875
.8 TO .9	.8125
.9 TO 1.0	.6875

If the battery voltage reaches 97% of its float voltage (corrected for temperature), it is automatically determined that the battery's state of charge is increased at one percent per second until that value is reached.

Battery capacity at temperature is estimated by an increase of 0.2% per degree C above 25 C, and reduced by 0.75% per degree C below 25 degrees centigrade.

If the battery voltage drops below the minimum allowed, it is automatically determined that the battery's real state of charge is lower than the estimate, and the estimaed state of charge is reduced at 1% per second until the voltage rises above minimum due to load shedding or until 0 state of charge.

For every ampere-hour of discharge by the battery, one equalization fraction ampere-hour of extra or equalization charging is automatically programmed when energy becomes available and 100% state of charge has been reached. This is needed to bring all cells up to full charge and to stir the acid by generating small amounts of gas. This prolongs total percentage to avoid excessive water useage.

If the battery goes through many partial charge and discharge cycles before equalizing, the total equalization charge is limited to the equalization total percentage to avoid excessive water useage.

During charging and equalization, the battery is allowed to rise to the equalization voltage. After equalization is completed, the battery voltage is limited to the float voltage, which is below the point of significant gassing and so uses up less water. This float level charge makes up for internal battery self-discharging.

The float and equalization voltages are equal to a nominal voltage per cell minus 0.22% per degree C above nominal (temperature in degrees C minus 25 C), or equal to the absolute maximum battery voltage, whichever is lower. The battery minimum voltage has the same temperature compensation and an absolute minimum value.

The result of these calculations, then, are the battery voltage maximum limit, the battery voltage minimum limit, the state of charge, the temperature corrected state of charge, and the equalization charge needed.

Battery state of charge is channel D00 and battery corrected state of charge is channel D40 on the LCD display and the serial port. These are the most frequently read values, and so corrected state of charge is put on the display by default at power up or after a data dump at half hour intervals. A diagram of the battery control algorithm is given in Figure 3.2.1-1.

3.2.2 Array Control

If the battery is not fully charged and equalized, the voltage limit is the equalization voltage. If fully equalized, the voltage limit is the float voltage. Until reaching voltage limit, the solar array charges the battery as much as the available energy allows.

There are two automatic modes of array control resident in the program. The first is discrete array control. This turns strings on, one per second as long as the battery voltage is less the 94% of its maximum limit. It makes no change until the battery voltage reaches 97% of its maximum limit. Between 97% and 100% of its maximum limit, strings are turned off one per second. Above 100% of maximum limit, all strings are turned off. The battery voltage used for this algorithm is averaged over four samples to minimize noise problems.

The second mode is maximum power tracking. The controller drives up to 10 power modules with a 20kHz pulse width modulated drive signal, 13 volts = high = OFF, 0 volts = ON. The pulse width is set in 1/4 microseconds increments. It is initialized to 0 and changed in small (1/4 microseconds) or large (2 microseconds) steps every 100 milliseconds. At each step, the total delivered power is calcualted and compared with the power delivered at the previous duty cycle. If the power changed by less than 1.5%, and if no limiting conditions were encountered, small steps are used, otherwise big steps are used to speed up the process.

If the power level was found to be constant or increasing, the next step is taken in the same direction as the last. If the power was found to be decreasing, the direction of step in duty cycle is reversed. Therefore, the duty cycle is adjusted to the maximum power operating point, and hunts there +/- one or two small steps, representing an operating point within +/- 1% of maximum power.

If the charger current limit or battery maximum voltage limit is reached or if switch S2 is open, the duty cycle is always decreased one step. This converts the controller into a constant voltage or constant current power supply, normally used for finishing charge of batteries or for motor starting without batteries.

Also, if the inhibit switch S1 is closed, the duty cycle is set to 0, turning off the solar array. Switch S2, normally closed, is usually used as a motor thermostat, while S1 is often used as a water tank float switch level control. A manual switch in parallel with S1, normally open, provides a manual array shut-off function.

BATTERY STATE OF CHARGE CONTROL ALGORITHM

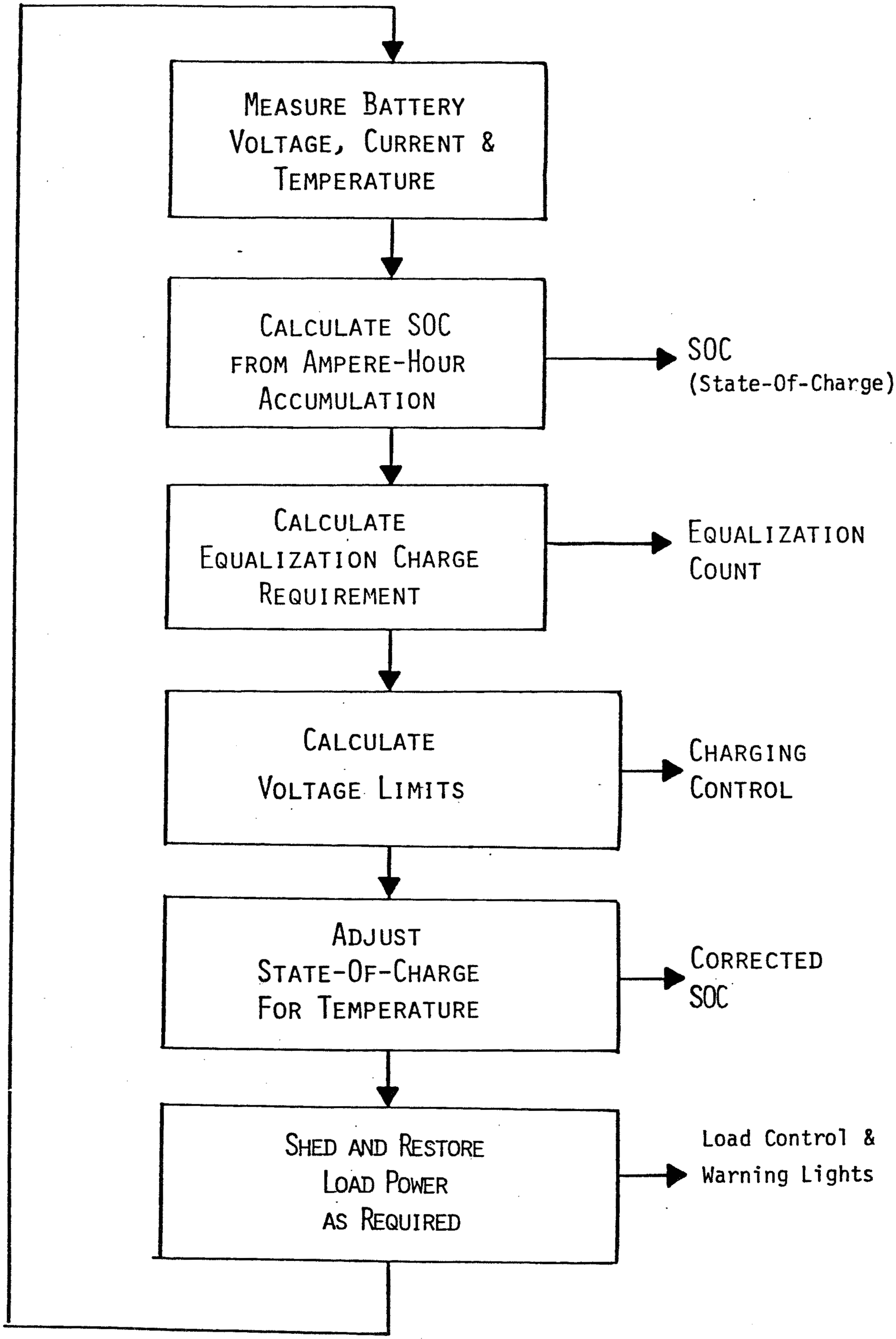


FIGURE 3.2.1-1

The sequence of events for max power tracking is as follows:

1. Pulse width is set on the previous program pass.
2. 12 milliseconds settling time elapses.
3. 16 values of each of the 10 charger string currents, the battery voltage, & the battery current are stored in an array. The table is filled by taking a sample of all values once every 4 milli-seconds, which generates a column of "instantaneous" data. After 64 milli-seconds, all the rows of data are full and average values are calculated.
4. User output power is calculated. In order to avoid start-up problems with low impedance (motor) loads, the actual parameter to be maximized is (total charger amps) X (battery volts + 100).
5. Old and new power are compared. If power is increased, the direction of pulse width step is left the same. If power is decreased, the direction is reversed.
6. The size of the pulse width change is calculated based on the size of the power level change. This is a novel feature which allows fast acquisition and accurate tracking of the max power points.
7. The pulse width is changed, and the process repeats at 100 millisecond intervals. This algorithm is diagrammed in Figure 3.2.2-1.

3.2.3 Load or Auxiliary Charger Control

The battery is assumed to be loaded by up to five separate load busses, with separate voltage sensing, current shunts in the negative load, and relay control responsive to the optoisolator output 1-5 of the controller. (Optoisolator ON=load ON.) These may in fact be auxiliary energy sources as well as loads.

Each load bus is automatically monitored for excessive current. If current exceeds 120 percent of the nominal value (50mV on the shunt), the load is turned OFF, the audio and red LED alarms turned ON, and manual reset is required via keyboard or serial port. Under normal conditions, each load bus has two corrected state of charge thresholds; one below which the load is turned OFF (shed) and one above which the load is turned ON (restored). By reversing the function of the optoisolator, an auxiliary generator can be properly controlled the same way. This provides fail-safe operation, since the optoisolator going off turns loads OFF and generators ON. Note: for either case, shed level is less than restore state of charge by an amount which avoids cyclic behavior. A list of sample control levels is given in Figure 3.2.3-1.

Each load or charger bus can also be manually controlled from the keypad or the serial port. To be turned ON, (Load ON, Charger OFF, Optoisolator ON), an optoisolator must be ON according to all automatic control algorithms and the manual control function. Otherwise, it is in the load OFF/charger ON condition.

It is often desirable to use more energy if it's going to be a sunny day. This may be to avoid short cycles of load useage or simply to increase energy useage efficiency. Therefore, between 8AM and

MAXIMUM POWER CONTROL ALGORITHM

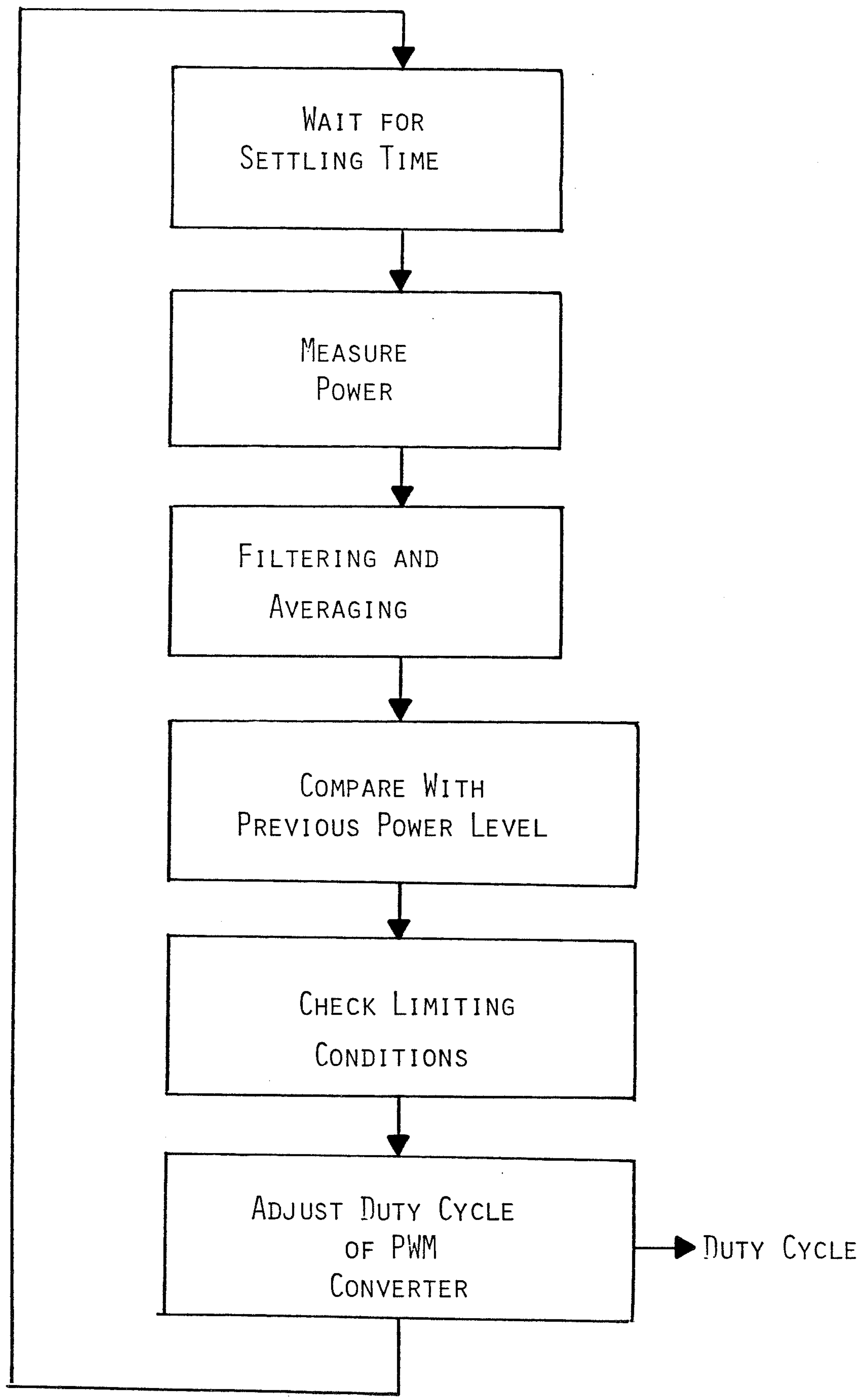


FIGURE 3.2.2-1

CORRECTED STATE OF CHARGE THRESHOLDS

(Percent)

LOAD	SHED	RESTORE
1	20	40
2	30	50
3	40	60
4	50	70
5	50	90
MORNING DELTA SOC	---	10
SOC TOP	---	80
EQUALIZATION FRACTION	---	20
EQUALIZATION TOTAL	---	40

ARRAY CONTROL MODE: Maximum Power Tracking

SWITCH CONTROL:

Charger ON if S1 open = 0 and S2 closed = 1.
Product storage path = Load Bus 6.

Auxiliary Equipment

Load Bus Shunts	50A = 50mV
Thermistors	25C = 10k ohms

FIGURE 3.2.3-1

12Noon, if the sunlight level is high enough to provide at least 10% of the maximum charger current limit to the battery, all the battery corrected SOC levels for load shed and restoration are decreased by a constant preset SOC, called DELTA SOC. This allows loads to turn ON earlier on a sunny morning than would otherwise be the case.

The MAC can perform a control strategy to allocate available energy into two different forms of storage. One of these is electrical energy stored in a battery. The other is a form of "product storage." This might be water stored in a tank, or thermal storage of cold in a refrigerator, freezer or ice stored in an icemaker.

To use this capability, it is necessary to be able to measure the amount of product stored or at least to indicate one of three product storage levels:

1. There is less than the critical minimum amount of product (water level or cold temperature in the freezer) and it has a high priority: PROD < PROD 1
2. There is an adequate amount of product, but more can be stored if energy is available. PROD 1 < PROD < PROD 2.
3. The product storage level is full (tank is full of water or refrigerator is at minimum temperature): PROD > PROD 2.

This product is assumed to be produced or pumped using energy from the same battery which is used to supply other electrical loads. To avoid over-discharge of the battery and to be sure the other loads get their share of energy, various state of charge (SOC) of the battery are defined:

1. Battery minimum charge: below this, the battery is below its minimum allowable state of charge and all load are off, above this non-product electrical loads are allowed: SOC1
2. The battery can be used to produce product but not other electrical loads above this point: SOC2
3. The battery can power both electrical loads and non-critical product levels above this point, but only non-product electrical loads below this point: SOC3 4. The battery will power all loads above this point: SOC4

This combination of trip points creates a set of four states of the controller, defined by the amount of product stored and the battery state of charge. This state diagram is shown in Figure 3.2.3-2. By setting various values for the boundary parameters, various priorities can be placed on the use of energy. To avoid limit cycles during operation, a buffer or hysteresis of specified amount is added to each boundary value when crossed in the increasing direction. The parameters for the product storage algorithm are given in the tables in Figure 3.2.3-3

PRODUCT STORAGE ALGORITHM

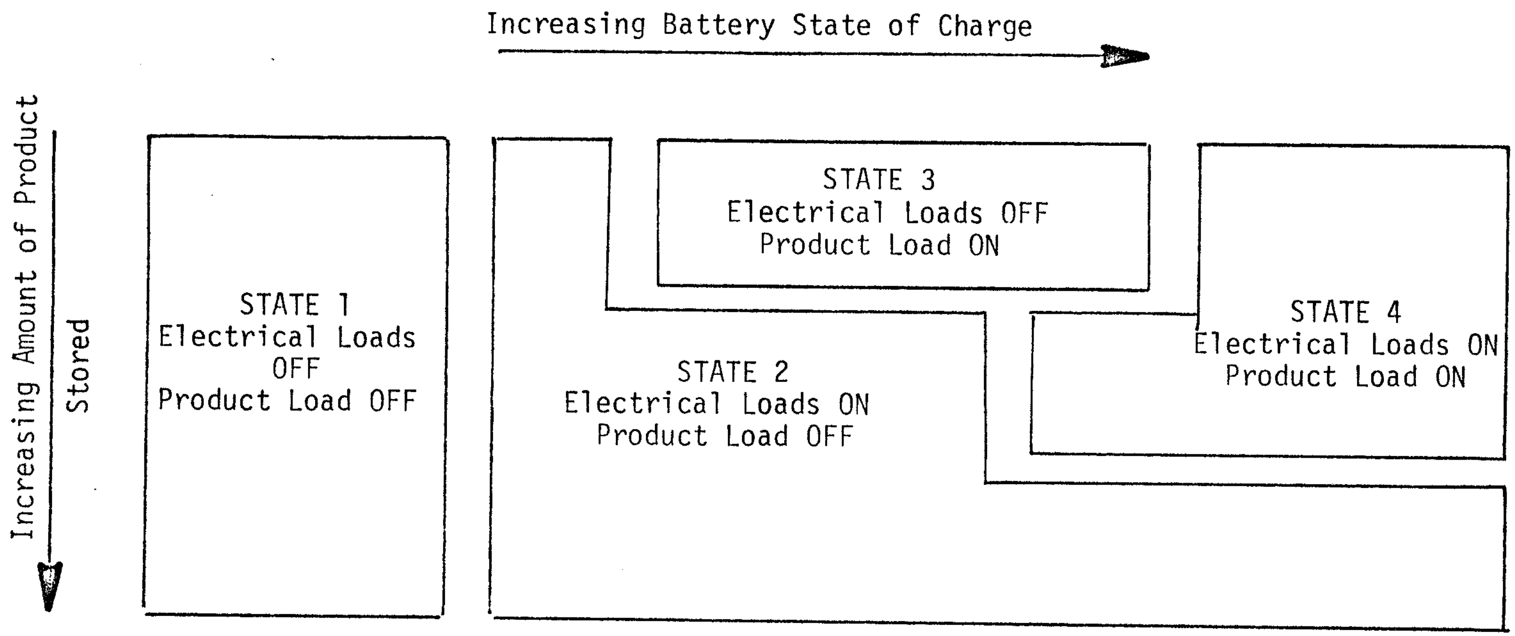


FIGURE 3.2.3-2

PRODUCT STORAGE ALGORITHM THRESHOLDS

(Percent)

PROD = (25 - T2) *3.3 T2 in degree C

DEFINITION	SYMBOL	VALUE	HYSTERESIS	VALUE
Elect loads above	SOC1	20	BUF SOC1	10
High Priority PROD only above	SOC2	50	BUF SOC2	20
Low Priority PROD above	SOC3	70	BUF SOC3	10
Low Priority PROD & Elect above	SOC4	70	BUF SOC4	10
High Priority PROD only til	PROD1	50	BUF PROD1	10
No PROD above	PROD2	80	BUF PROD2	10

FIGURE 3.2.3-3

3.2.4 Control Panel Functions

The control panel has a keypad for manual inputs, a 4-1/2 digit LCD display for selected quantities, a set of red, yellow, and green LED's for quick status summary, and an audio alarm. There is also an emergency switch located at the bottom of the unit which, when turned OFF, will turn OFF all array strings and loads. When turned ON, it will initialize the controller.

The audio alarm will sound if any "circuit breaker" function (overcurrent on a load bus) is activated, or if an array fault is detected.

The red LED will light if any circuit breaker function is tripped or if bus NO.1 has been shed due to low state of charge (extreme low battery). The LCD "Low Battery" warning will also turn ON.

The yellow LED will light is load bus NO.5 has been shed due to low state of charge, but load bus NO.1 has not. This is intended to signal that energy is in short supply, but not an emergency. (Lower number loads usually have higher priority.)

The green LED is lit as long as the program is running. If a watchdog timer is not reset by the program, this LED goes out and the array is turned OFF.

The LCD display can be thought of as a multimeter which will display one of a menu of system parameters. The default condition is state of charge, and after a data dump each half hour, it returns to this parameter. It can also display any array string voltage, any charger output string current, (not the array current, but the converted output from that string at the battery voltage level), the delivered string power to the battery, the battery voltage current or power, any of five load bus voltages or currents or powers, the battery temperature, one other temperature (used here for freezer product measurement), the pulse width, the equalization charge needed, the system zero voltage and 4.00 volt reference level, the temperature corrected state of charge, the time of day, and the software version number. See the attached signal channel list. Enter the desired channel code (a letter, two numbers, and a #) for display. The # sign has the effect of an "enter" command. The * sign cancels an incomplete entry. A list of public acess functions is given in Figure 3.2.4-1. In addition, after entry of a password on the keyboard, an additional set of protected functions becomes available, listed in Figure 3.2.4-2.

3.2.5 Self-testing

The keypad can also be used to change control functions. To do this, the correct password must be entered: 4 digits followed by a #. Password accessible functions are summarized on the attached command list. These include setting a new load and restore thresholds for each load bus, setting initial state of charge and setting the manual control for each load bus optoisolator to "ON = 1" or "OFF = 0".

PUBLIC FUNCTIONS

SEQUENCE	FUNCTION
*	Clear Function
aaaa#	Activate password-accessible functions if user password matches (see below)
Ann#	Read channel nn voltage (mult)
Bnn#	Read channel nn current (mult)
Cnn#	Read channel nn power (mult)
Dnn#	Read misc data channels (mult)
AA#	Display software version number
BB#	Read time, hours and minutes (mult)
CC#	Initiate "dump" of machine state to serial port
DD#	not used

FIGURE 3.2.4-1

PASSWORD ACCESSIBLE FUNCTIONS

SEQUENCE		FUNCTION
Anmmm#		Set load shed threshold for load n at mmm
Bnmmm#		Set load restore threshold for load n at mmm
Cmmm#		Set the initial percentage SOC at mmm%
Dnnm#		Set device nn to condition m, where m must be either a "1" (ON) or a "0" (OFF)
	device 0	Audible alarm
	device 1-6	User load requests 1 through 6
	device 7-11	Overload trip resets for loads 1 through 5
	device 12-17	PWM buffer #1 controls 1 through 6
	device 18-23	PWM buffer #2 controls 1 through 6
	device 24-25	Yellow and Red LEDs
AA#		Initiate lamp and annunciator test (accessable only in test/cal mode)
BB#		Toggle from run to test/cal mode (system comes up in the run mode)
CChhmm#		Set time, hours and minutes
DD#		Cancel password authorization

FIGURE 3.2.4-2

Also, the unit can be placed in a test/cal mode where data is refreshed but the states of charge, max power tracking and load controls do not automatically change. This allows them to be manually set and measurements taken. For example, a fixed pulse width can be set from the serial port, or a lamp test can be performed. Note that this mode is not well protected and the system should not be left in this condition unattended.

A lamp and audio alarm test can be performed in this mode. This lights all LEDs, sounds the alarm and put test digits in the display. The keypad in password-access mode can also be used to set the time in hours and minutes on a 24-hour clock and can cancel the password-access mode.

The display performs one more automatic test. If one of the charger power module output currents is less than a constant (1 amp) below the average of all such currents, the "Continuity" segments of the LCD is turned ON. This means that one string of the array or one power module of the charger is either shadowed or broken. Examination of array voltages and currents will show the cause.

Besides these external tests, the unit performs an automatic test of its RAM and ROM access whenever it is started up. When running, any error causing the program not to execute properly will result in a watchdog timer fault turning off all loads and the solar array and causing the green LED to go out.

3.2.6 Serial Port Functions

The manual test/cal mode was partially described in the previous section. However, with an RS232C terminal, much more extensive manual testing is possible. A standard debug monitor is implemented which can examine memory locations, insert bytes into memory, begin execution at a given memory location, insert or remove breakpoints, and download a program into memory.

In addition, any of the keyboard command functions can be performed from the serial port. A list of these commands, plus the debug monitor commands is in the serial port section of the command list. Note that no password is requried via the serial port. The debug and monitor functions are listed in Figure 3.2.6-1, and the maintenance functions are listed in Figure 3.2.6-2.

A data dump of all system voltages, currents, temperatures, switch positions and states of charge is sent to the serial port every half hour or whenever requested. A printer there will provide data logging. The format of this data is given in Figure 3.2.6-3.

This powerful access allows many possible uses. The processor can be automatically tested during manufacture. A printer attached at the site can serve as a data logger. A telephone modem or other communication interface would allow automatic remote data logging or even, remote system control. A tape recorder will allow post-processing of recorded data.

DEBUG MONITOR FUNCTIONS

SEQUENCE	FUNCTION
^H, backspace, del	Deletes last character entered. Echoes backspace, space, backspace to allow overwriting the last character entered when a CRT terminal is used
^U	Causes CPU to ignore present command line
^Z	Return to command mode
M addr	Opens a memory location at the specified address (requires 4 hexadecimal digits). Successive "" (space) characters increment through memory, while "-" characters decrement through memory. At any time the contents of a location may be altered by entering the new data followed by a carriage return.
G addr	Begins execution at the specified address. If no address is specified, execution begins at the present PC location
B addr	Places a breakpoint at the specified address. This trace mode will only work on code located in RAM
X	Removes existing breakpoint
l addr	This permits a program to be downloaded from a host machine to memory starting at the specified address.
F start_addr end addr datum	Fills the specified memory range with specified byte of data.

MAINTENANCE and LOGGING FUNCTIONS

SEQUENCE	FUNCTION
W pwm timer no. duty cycle	Set the Power Module duty cycle to specified value. If duty cycle max pwm, the default duty cycle is set to max pwm.
O digit no. value	Display the value in the specified digit on the LCD display.
Enn	Read channel nn voltage
Inn	Read channel nn current
Pnn	Read channel nn power
Dnn	Read misc data channels
Qnn	Query channel nn for "raw" A/D data

FIGURE 3.2.6-2

DATA FORMAT

TIME: hh:mm

E00 = xxx.x VOLTS	I00 = xxx.x AMPS
E01 = xxx.x VOLTS	I01 = xx.xx AMPS
E02 = xxx.x VOLTS	I02 = xx.xx AMPS
E03 = xxx.x VOLTS	I03 = xx.xx AMPS
E04 = xxx.x VOLTS	I04 = xx.xx AMPS
E05 = xxx.x VOLTS	I05 = xx.xx AMPS
E06 = xxx.x VOLTS	I06 = xx.xx AMPS
E07 = xxx.x VOLTS	I07 = xx.xx AMPS
E08 = xxx.x VOLTS	I08 = xx.xx AMPS
E09 = xxx.x VOLTS	I09 = xx.xx AMPS
E10 = xxx.x VOLTS	I10 = xx.xx AMPS
E31 = xxx.x VOLTS	I31 = x.xxx AMPS
E32 = xxx.x VOLTS	I32 = x.xxx AMPS
E33 = xxx.x VOLTS	I33 = x.xxx AMPS
E34 = xxx.x VOLTS	I34 = x.xxx AMPS
E35 = xxx.x VOLTS	I35 = x.xxx AMPS
D36 = xxx.x DEG C	D37 = xxx.x DEG C
E38 = x.xxx VOLTS	I39 = x.xxx VOLTS
D00 = xxx%	D40 = xxx%
D41 = xxx%	D42 = xxx%

S1-1	S2-0		L1-1	L2-0	L3-1	L4-0
			L5-1	L6-0		
A1-1	A2-1	A3-1	B1-1	B2-1	B3-1	B4-1
A4-1	A5-1	A6-1	B5-1	B6-1		

TOT CHGR I = xxx.x AMPS

NOTE: 1=ON, 0=OFF

The use of a small RS232C terminal by a repair technician allows fault isolation and diagnosis beyond the capabilities of the keypad and display, since small special test routines can be run. Software modification is greatly eased as well.

3.3 SYSTEM CONFIGURATION

To utilize these rather sophisticated control algorithms is very simple. Typical system configurations for a pumping unit without batteries is shown in Figure 3.3-1. Note that this is just the minimal amount of wiring required for such a system. Similarly, the system configuration for a battery-type system is shown in Figure 3.3-2. The processor contains all the control complexity, making the system designer's job easier. The functions performed by the processor are summarized in Figure 3.3-3. To implement these, only the proper system parameters need be inserted in the processor EPROM memory. In this way, the user or installer needs the minimum knowledge of system theory to be able to use the controller.

4.0 PRODUCTION COST ANALYSIS

4.1 MANUFACTURUNG COST

The results of costing the system can be summarized by several points. Each extra analog channel to be measured is expensive because the filtering and multiplexing associated with each channel quickly adds 3 to 5 dollars per channel to the overall cost. The analog input section of the processor represents almost half the PC board area and about 1/3 of the cost.

The largest single area of cost is the CPU itself, dominated by memory cost, mainly from EPROM's. These are rapidly becoming less expensive, and in a few years, a single larger EPROM and a single 2K RAM will probably enable the entire system to be built at lower cost and less power.

The power supply, rather expensive at present, can be simplified for most applications in the future, since the total controller power can be reduced. Also, a separate +12V output from the flyback stage is not needed, and -5V power can be derived from a Zener diode operated from the -12V supply. This cuts its cost by perhaps 1/3.

Availibility of a CMOS timer chip with the capabilities of the 9513 would help a great deal in reducing power and perhaps costs. In general, the increasing industrial use of CMOS is expected to bring costs of many components down. The power handling section of the controller, consisting of DC to DC power conversion modules, can be replaced in low cost battery systems with electromechanical or mercury displacement relays or solid state DC switches and the discrete array control algorithm. It will also cost half as much. This will be the direction of choice for battery charging as solar arrays become cheaper. Also, eliminating these PWM outputs would allow replacement of the 9513 timer chip with a lower power, less expensive timer. However, for many motor drive applications, the

TYPICAL PUMPING SYSTEM WIRING

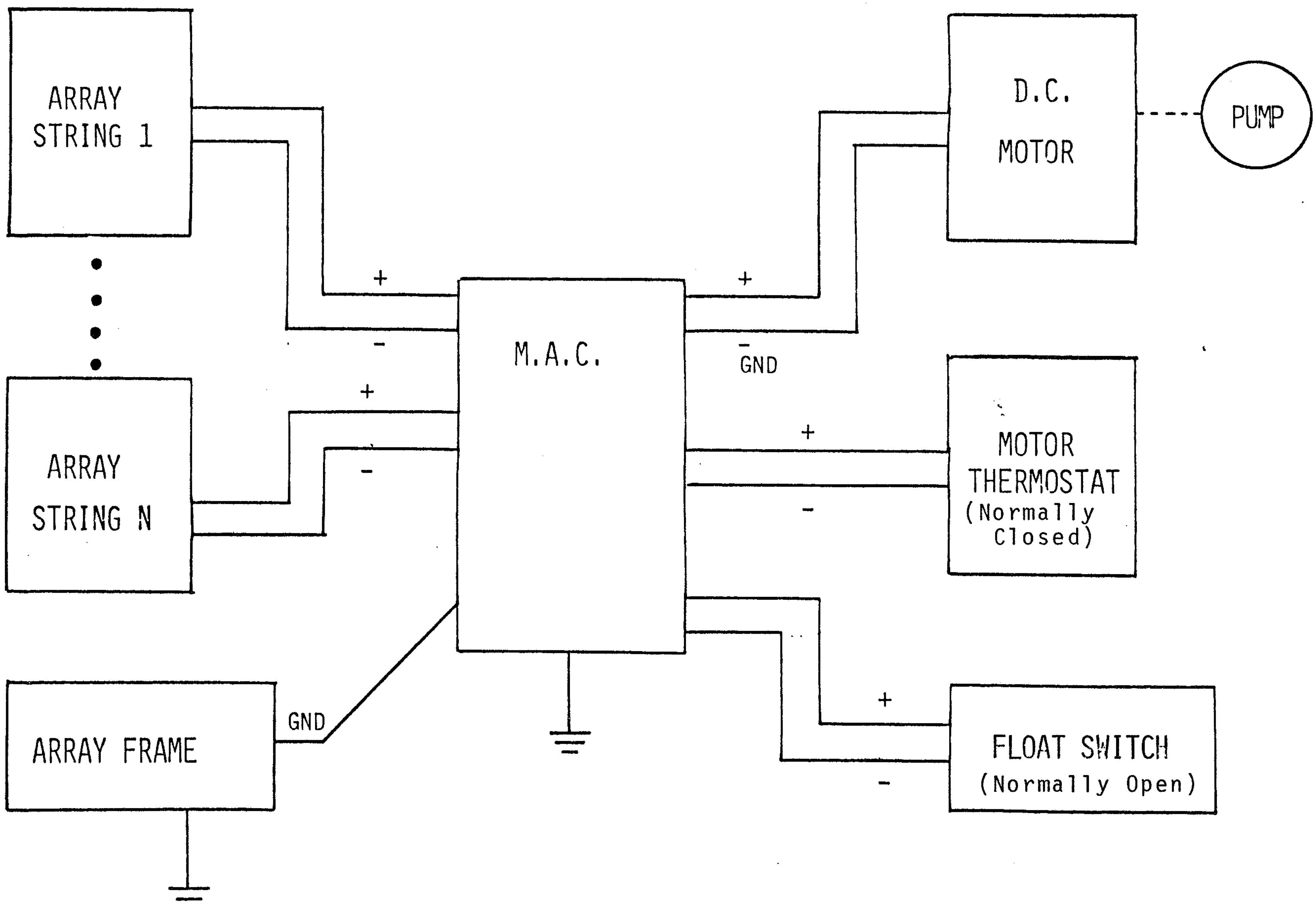


Figure 3.3-1

TYPICAL BATTERY SYSTEM WIRING

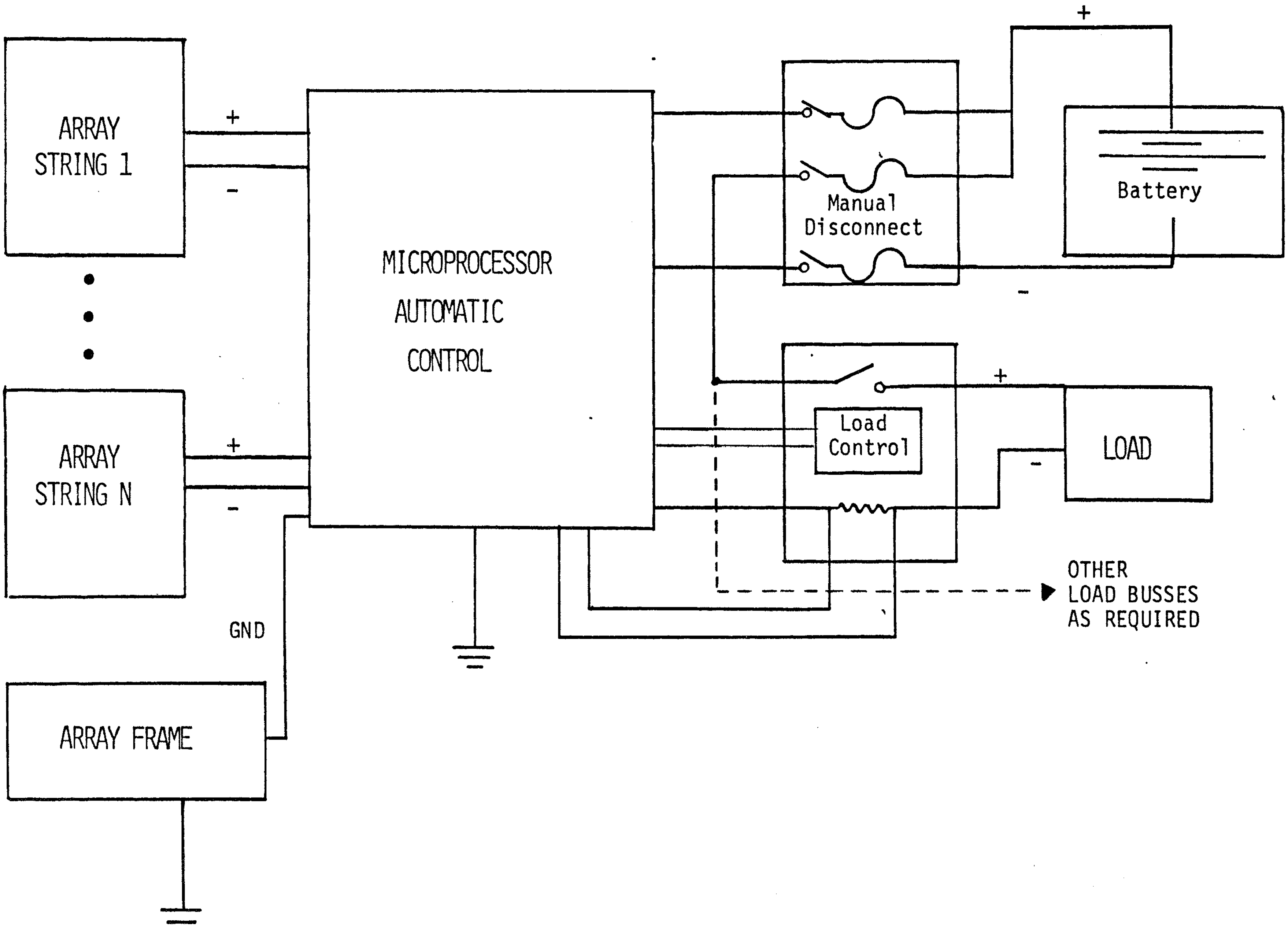


FIGURE 3.3-2

MICROPROCESSOR CONTROL FUNCTIONS

1. BATTERY CONTROL
   - A. State-Of-Charge Estimation and Correction
   - B. Equalization Charging
   - C. SOC Initialization and Enunciation
   - D. Warning Lights

2. PHOTOVOLTAIC ARRAY CONTROL
   - A. Discrete Subarray Switching
   - B. Maximum Power Point Tracking

3. LOAD MANAGEMENT
   - A. Priority Load Sector Management
   - B. Backup Generator Control
   - C. Manual Load Control
   - D. Product Storage Energy Allocation

4. SYSTEM STATUS
   - A. DC Voltages, Currents, Power Levels
   - B. Temperatures
   - C. Battery SOC, Corrected SOC
   - D. Variable Control Level
   - E. Relay Status

5. AUTOMATIC TESTING
   - A. Processor Self-Test
   - B. Load Circuit Breaker Functions
   - C. Array Continuity Test
   - D. Battery Level
   - E. Audible and Visual Alarms

6. DATA LOGGING AND COMMUNICATIONS
   - A. System Status Log Each 30 Minutes
   - B. Remote Telemetry
   - C. Remote Control
   - D. RS 232 Port

MPC arrangement is superior.

The power module is the only proprietary element in the controller. It is a standard product of TriSolarCorp.

A material cost breakdown of the controller subsystems is given below for present day quantities of 10:

DESCRIPTION		COST
Current Measurement Mux	$	34.04
Analog Mux		56.69
A/D Converter		27.83
PWM Buffers and Array Control		24.75
CPU		108.29
Power Supply		83.14
Display and Interface Board		49.82
Resistor and Interface Board		5.00
Processor Subtotal		499.56

The resulting material costs are then calculated below for the 1KW, 5KW, and 15 KW systems:

	1KW No MPC	5KW MPC	15KW MPC
PROCESSOR	459.56	499.56	559.56
POWER SECTIONS	100.00	1121.20	3363.60
ENCLOSURE	165.00	165.00	495.00
TERMINALS	20.00	27.00	81.00
HEAT SINKS	0.00	138.00	416.64
SHUNTS AND HARDWARE	36.25	36.25	68.75
TOTAL MATERIAL	780.81	1987.89	4984.55

Material prices are assumed to be reduced by 25% for each 10X increase in quantity.

Labor costs are very volume dependent. Use of automatic test equipment, wave soldering or automatic component insertion are examples of labor-saving techniques which can be applied at various levels of production.

Total labor costs for various models and production levels are estimated below:

MODEL QUANTITY/YEAR	1KW	5KW	15KW
10	476	768	1400
100	280	452	819
1,000	168	266	482
10,000	100	160	282
100,000	60	100	170

Labor is broken down into wage levels. For example, for each model given, the following hours are needed for quantities of 1,000 per year:

	$/hr	1KW	5KW	15KW
Class 1	22.60	1	2	3
Class 2	13.50	3	6	10
Class 3	7.00	15	20	40

These labor estimates are conservative; the actual number for such volume of production will probably be lower. However, these seem to be reasonable based on our limited experience with the protopype unit.

Resulting total direct costs per 5KW unit, including labor and 100% overhead plus material are therefore just under $2400 in quantities of 100 per year. The 1KW unit in the same quantities costs under $1200. These are quite reasonable costs compared to alternative custom designed controllers. The real payoff, of course, occurs at higher volumes.

The total direct cost of the three models, including material, labor, and 100% overhead is summarized in the following table.

QUANTITY/YR	1KW	5KW	15KW
10	1732	3524	7785
100	1145	2395	5377
1000	775	1650	3768
10000	529	1158	2667
100000	367	829	1916

4.2 MODELLING OF INDUSTRIAL PRODUCER

In order to capture all the costs associated with producing this controller in a rapidly growing commercial environment, a model was constructed of a company which would make only this unit in quantities of 10; 100; 1,000; 10,000; and 100,000 per year in successive years. Industry standard or typical ratios were used for the computer electronics industry in Massachusetts, based on advise of consultants experienced in this area. Overhead, inventory, equipment, G&A, and other costs were captured in this model. The result gives a picture of cash flow and return on investment as well as profitibility.

The costs of the controller are used for this hypothetical company producing various quantities of 1, 5, and 15kW controllers, the resulting income statement, balance sheet, and cash flow projections apply. Note that for quantities under 1000/year, a dedicated company for just this purpose is not feasible, and losses result. For very small numbers of units (10) per year, costs go up and prices reflect the custom nature of the product. However, for larger volumes, prices are quite reasonable and return on investment becomes very attractive. Figure 4.2-1 (3 pages) shows these figures.

The first three lines show the unit sale price of each of the three controllers: 1=1kW, 2=5kW, 3=15kW. The next three lines, labelled unit sales, show projected sales numbers for each type. The following three lines, labelled number of people, give the number of employees needed at each of three salary levels, defined on three wage level lines immediately below. The following three material cost lines give cost of material per unit for each of the three types of controller. The last line, micro dollars per unit, indicates the equivalent material cost of the PV panel needed to power the controller. The interest rate for each year is indicated below.

The next section is an income statement for the model company. Sales and cost of sales are computed from the above values. G and A and overhead are calculated using typical electronics industry ratios. The resulting net after tax shos profitability after sales pass a few million annually, and look good at large volumes as a fraction of sales.

The next lines are a balance sheet showing assets and liabilities. Typical industry ratios for inventory, receivables, and payables are used. The line marked "plug" represents extra capitalization needed, loan or equity, to fund this rate of rapid expansion.

The last page shows cash flow, and points up the negative cash flow from operations due to the rapid growth rate. The use of investment cash is divided mainly between inventory increase and purchase of fixed assets. The return on investment can be seen by company increase in long term debt with net after tax profits.

TSCx:" VC CENTR TRISOLARCORF

ORIGINAL PAGE IS OF POOR QUALITY

BALANCE SHEET AND INCOME STATEMENT SUMMARY

	--1983--	--1984--	--1985--	--1986--	--1987--
SALES PRICE 1	1700	1600	1400	1200	1000
2	9000	8000	5000	3000	2208
3	20000	18000	10000	7000	5000
UNIT SALES 1	2	60	600	6000	60008
2	8	35	350	3500	35000
3	0	5	50	500	5008
# OF PEOPLE LEVEL1	1	1	5	15	100
LEVEL2	2	2	8	30	258
LEVEL3	3	5	18	90	800
WAGE LEVEL1 47000	47000	47000	47000	47000	47000
2 28000	28000	28000	28000	28000	28000
3 14500	14500	14500	14500	14500	14508
MATERIAL COST1	781	586	439	329	247
2	1988	1491	1118	839	629
3	4985	3739	2804	2103	1577
MICRO $ PER UNIT	50	40	30	20	18
INTEREST RATE	0	.18	.17	.16	.16
–					
SALES	75400	466000	3090000	21200000	162000000
COST OF SALES	196696	324134	1703578	9640836	74154770
GROSS MARGIN	-121296	141866	1386422	11559164	87845230
G&A	24128	149120	797220	4409600	25596000
OTHER INCOME		-47746	-60273	-139819	-1008203
NET INCOME	-145424	-54999	528928	7009745	61241027
NET AFTER TAX	-145424	-54999	371246	3645068	31845334
CURRENT ASSETS					

CASH	25000	25000	25000	25000	25000
ACCOUNTS RECEIVABL	75400	70606	561818	5170732	39512195
DEPOSITS					
INVENTORY	75400	90037	608421	4820418	37077385
OTHER					
TOTAL	175800	185643	1195239	10016150	76614580

FIXED ASSETS

FIGURE 4.2-1 A

TEST&SHOP	4565	30065	216589	2304348	17608696
OFFICE EQUIPMENT	4865	30065	72196	768116	5869565
DEPRECIATION	1459	9019	12629	51035	344513
OTHER LESS AMORT					
TOTAL	8270	51110	276156	3021429	23133748
TOTAL ASSETS	184070	236753	1471395	13037578	99748328

CURRENT LIABILITY					

ACCOUNTS PAYABLE	30371	39064	241309	1447171	11072064
NOTE PAYABLE					
TAX WITHHELD					
ACCRUED EXPENSES &	33869	43563	185396	1473246	11271561
PROGRESS BILLINGS					
TOTAL	64240	82627	426704	2920418	22343625
LONG TERM DEBT					

NOTES					
TOTAL	0	0	0	0	0
TOTAL LIABILITIES	64240	82627	426704	2920418	22343625

EQUITY					

COMMON STOCK					
ADD PAID IN CAPITL					
RETAINED EARNINGS	-145424	-200423	170823	3815890	35661224
TOTAL EQUITY	-145424	-200423	170823	3815890	35661224

EQUITY & LIABILTYS	-81184	-117797	597527	6736308	58004850

PLUG	265253	354549	873868	6301271	41743478
SOURSES OF CASH					

SALES		466000	3090000	21200000	162000000

FIGURE 4.2-1 B

[illegible] REC INCREASE	[illegible]	[illegible]	[illegible]	[illegible]
TOTAL	470794	2598788	16591086	12765
EXPENSES	520999	2718754	17554932	13015
LESS NONCASH EXPEN	-7560	-3610	-38406	-29
CHG DEPOSITES	0	0	0	
CHG INVENTORY	14637	518384	4211997	3225
CHG OTHER	0	0	0	
CHG ACCT PAYABLE	-8693	-202245	-1205862	-962
CHG NOTE PAYABLE	0	0	0	
CHG WITHHOLDING	0	0	0	
CHG ACCRUED EXPENS	-9694	-141833	-1287851	-979
CHG PROGRESS BILL	0	0	0	
TOTAL OP CASH USE	509690	2889450	19234811	14269
CASH FROM OPER	-38896	-290662	-2643724	-1503
SALE OF FIXED ASST				
INCR LNG TERM DEBT	89296	519318	5427403	3544
CHG STOCK &PD IN C	0	0	0	
TOTAL SOURCES	50400	228656	2783679	2040

USES OF CASH				

PURCHASE FIXED ASS	50400	228656	2783679	2040
DECREASE LNG TM DT				
TOTAL USES	50400	228656	2783679	2040
INC/DEC IN CASH	0	0	0	

ORIGINAL PAGE IS OF POOR QUALITY

FIGURE 4.2-1 C

5.0 TESTING

The MAC 5 kilowatt prototype was tested in several stages. Its basic hardware was tested at TriSolarCorp. Its software was tested at the Mellon Institute. Then the integrated hardware and software was extensively tested at TriSolarCorp, resulting in further development of some algorithms. Finally, testing in four system configurations was planned for NASA's STF system test and completed at TriSolarCorp The test facility included a 2.4kW power supply, a 480W solar PV array, a 100A-hr 108V lead acid battery bank, a 5kW resistive adjustable load, a 1HP DC motor driving a jack pump with adjustable hydraulic load, and assorted control and sense elements. A schematic of the test set-up is included in Appendix A. The first set of tests used power supplies to calibrate the various inputs and to simulate battery charge and discharge cycles. These tests showed that all systems worked properly, with accuracy within the +/- 2% specification. It became clear that accuracy better than 1% could be attained by replacing the discreet resistors on the divider board with precision resistor networks, a change involving very little extra cost. During this test phase, a subtle program bug in the battery state of charge subroutine was found and fixed. Power module efficiency was verified to be 97 to 98% at these voltages. The second test phase used the controller with the battery, solar array, and resistive loads.

The battery charge control algorithm works as described. Its estimate of the available capacity is limited mainly by the accuracy of the battery's rated capadity at a given discharge rate. This figure is reproduceable only to about 10% accuracy, and is usually based on 5 years or fixed cycle life end of life capacity. The initial capacity increases for a few cycles then slowly decreases with life. Therefore, the displayed battery capacity and the load management algorithm tend to be conservative by 10 to 20% over the first years of battery life. Long term low current operation is limited by internal battery self-discharge, typically 1% to 2% per week, and current effect errors of 0.1% of maximum, which is often a similar percentage per week. This is corrected whenever the battery is fully charged, so that batteries cycled daily or weekly are not affected.

The array control modes, both max power tracking and discrete switching, work very well and provide a nice tapered current finishing charge for a battery with very low water loss.

Load management in response to battery state of charge is very good. It is free of the chattering and instability characteristics of many voltage-related load management schemes.

Instrumentation of the system using the keypad and LCD display is very effective. A permanently posted list of command codes on the unit near the keypad was found to be useful.

Data logging via the RS232C terminal to an inexpensive printer will be very helpful in village systems. Of less use in smaller applications, it is quite helpful for maintenance or fault diagnosis using a small hand-held battery powered RS232 terminal. The possibilities inherent in a phone coupler or other communications port for remote system control have not been explored, but they are potentially very interesting. The self-test functions of the controller are an effective means of allowing unskilled personnel to monitor a complex PV system.

The third set of tests used the MAC to operate a jack pump without batteries. Combinations of insolation (simulated by moving the array), total dynamic head, and load-levelling flywheel size were compared. The controller successfully performed its maximum power tracking task here as with a battery load, unless the variation in load current was faster than a fraction of a second over a range of over 50% change in load current. The MAC was also compared to previous models of max power trackers. The max power tracking with two steps sizes is a real advance in the state of the art, as it provides a combination of more accurate tracking of a static max power point load and faster acquisition time for a varying load or insolation. A chart comparing these performance figures is given here. Note that this control method is proprietary and a patent has been applied for.

COMPARISON OF MPC FUNCTIONS

	PREVIOUS STATE OF THE ART EQUIPMENT	MAC
Acquisition Time	3 sec	1 sec
Max Dither, stable	3%	1%
Product	9	1

This means, for example, a volumetric pump or other slowly cycling loads can be dynamically matched to the solar array with reduced need for expensive or costly load levelling mechanisms such as large flywheels, batteries, etc. This opens up additional applications for the controller. Flywheels previously sized for 5 strokes of energy for the pump at full speed can now be made to store only 1 stroke of the pump at full speed, saving considerable cost and energy.

In fact, operating the jack pump without any flywheel resulted in no loss of output. However, system parameters were poorly controlled under this condition, and the array and motor voltage varied for short periods of time over a wide range.

In very low light levels (dawn or dusk) the operation of the controller was irregular as it alternately started and stopped internal operation. This is common with many solar devices, and it caused no damage or loss of output, but it is not well controlled. A change of control parameters is suggested to improve this: reducing the step size at small pulse widths, and reducing the minimum pulse width.

The last set of tests used the battery, array, power supply, resistive load, and simulated product storage to test the product storage dual-use algorithm. This operated as it should, but showed sensitivity to the programmed threshold values and was difficult to diagnose. It may be a bit complicated for remote village use, although it can be used to allocate energy between several uses. Its sophistication makes it difficult for a user to verify that it is operating properly, and this may make it less popular than other approaches based solely on battery status.

The overall result of the testing was that the MAC unit works and works very well. In comparison with other existing controllers, the microprocessor automatic controller worked as well as, or better than, any existing PV system control equipment, and can be adapted to a wide variety of systems simply by plugging in an EPROM. This makes it a very attractive controller for PV systems where large size (over 2kW), remote location or special control requirements justify this type of unit.

6.0 CONCLUSIONS

The battery charge control algorithms in the MAC provide stable, predictable operation of the system and will do a good job of extending battery life, minimizing maintenance, and accurately estimating state of charge. The load management controls are flexible and provide good prioritized control without any problems of chatter or noise sensitivity.

The array controls work well in either the maxpower tracking, pulse width modulated mode or in the discrete array switching mode with sequential operation. In the maxpower tracking mode, the dual step size algorithm is a real advance in the state of the art and allows tracking of varying loads which is both faster and more accurate than previous controllers.

The metering and data logging functions provide a good picture of overall system status. The self-test functions auch as array string continuity testing will be very useful in substituting for skilled operators at remote sites and allowing scheduled system maintenance.

The potential for remote system control and monitoring is considerable. The full use of the RS232 port capabilities of this controller have not yet been fully explored. Remote system monitoring or control, various types of data logging, or other maintenance or communications functions are possible. This may make the microprocessor controller attractive in other kinds of systems, such as utility interactive PV systems or non-PV applications.

The success of this development will allow rapid production of the unit for field application.

APPENDIX I

SCHEMATIC DIAGRAM

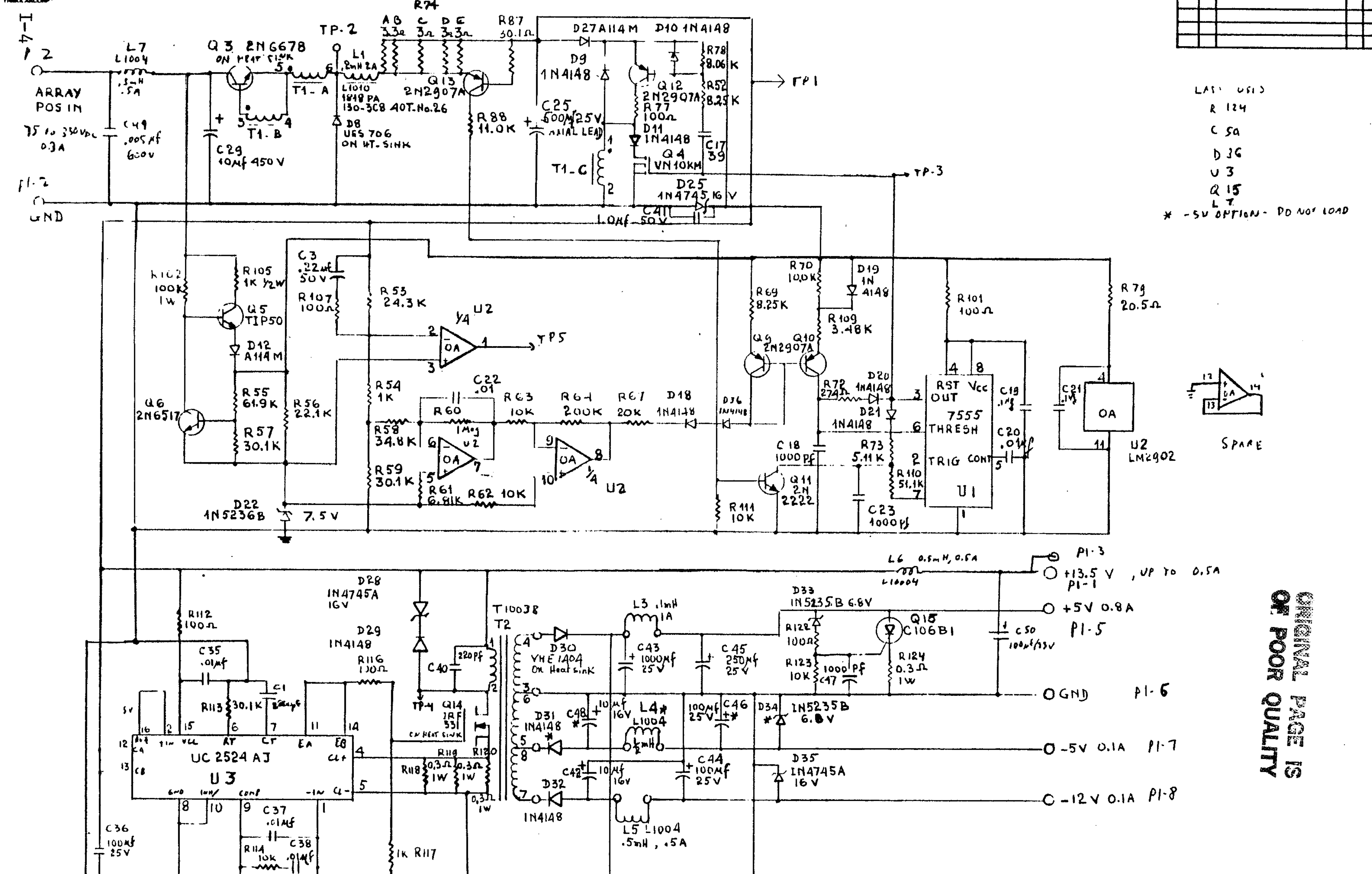
APPENDIX I

The following drawings show the microprocessor automatic PV system controller at the system level, P10 (5 kilowatt) box level, and the board level.

The drawings attached include:

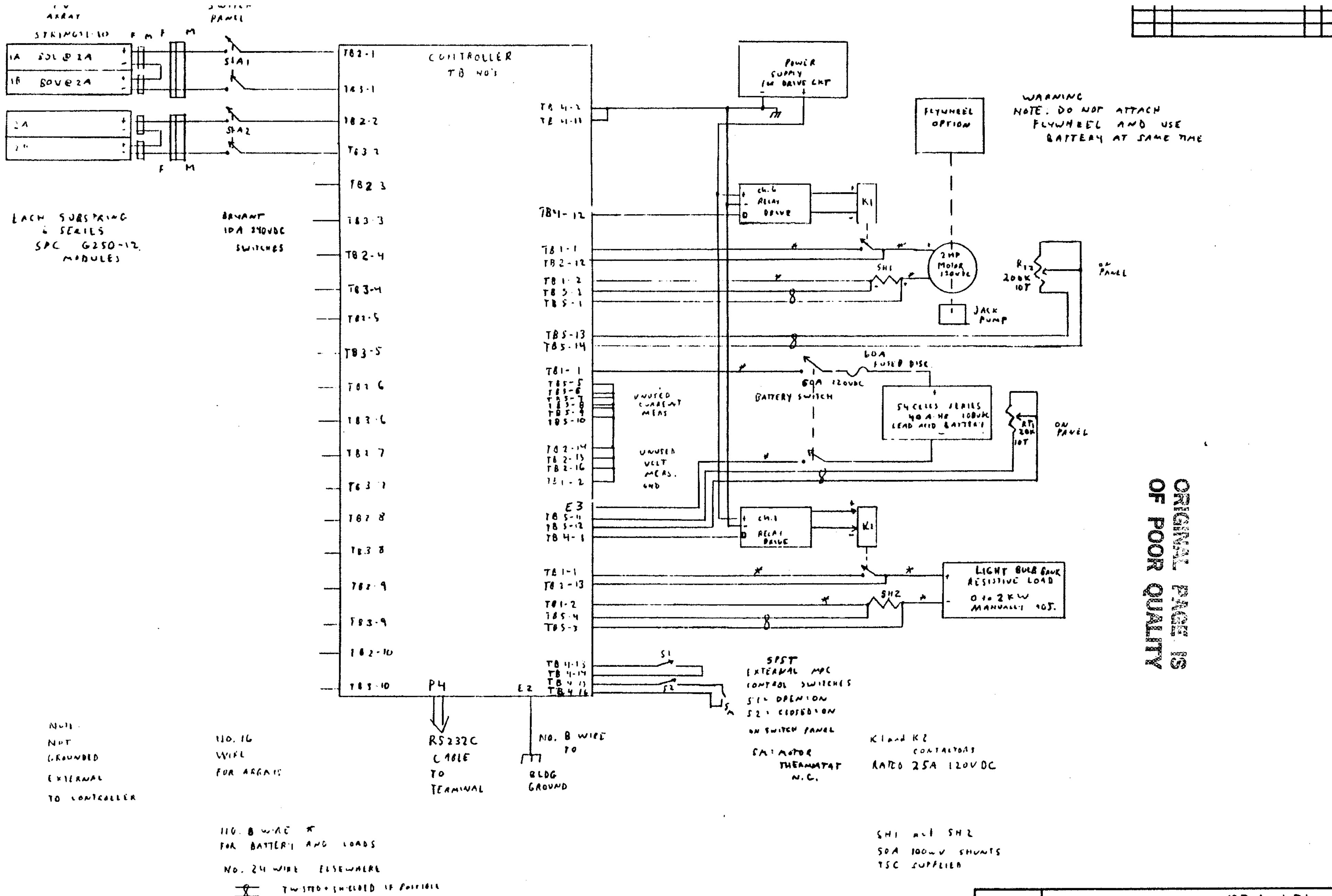
C02-00414-01	Microprocessor Power Supply
C02-00416-00	Microprocessor Controller Test Configuration
C02-00418-00	Microprocessor Display and Control Logic
D02-00413-01	Microprocessor P10 Wiring Diagram - Busses
D02-00413-02	Microprocessor P10 Wiring Diagram - Single Wires
D02-00420-00 Sheet 1	Microprocessor Schematic - Central Processor
D02-00420-00 Sheet 2	Microprocessor Schematic - PWM Buffers and Array Controls
D02-00420-00 Sheet 3	Microprocessor Schematic - A/D and Interface Logic
D02-00420-00 Sheet 4	Microprocessor Schematic - Analog Multiplexer
D02-00420-00 Sheet 5	Microprocessor Schematic - Current Measurement Mux

DATE	SYM	REVISION RECORD	DR	CK
11-21	A	L6, C50, D33 - Q15		



ORIGINAL PAGE IS OF POOR QUALITY

TOLERANCES EXCEPT AS NOTED	TriSolarCorp	10 De Angelo Drive Bedford, MA 01730
DECIMAL .XXX .X .005	MATERIAL	SCALE
	DRAWN BY TT	APPROVED BY ARM

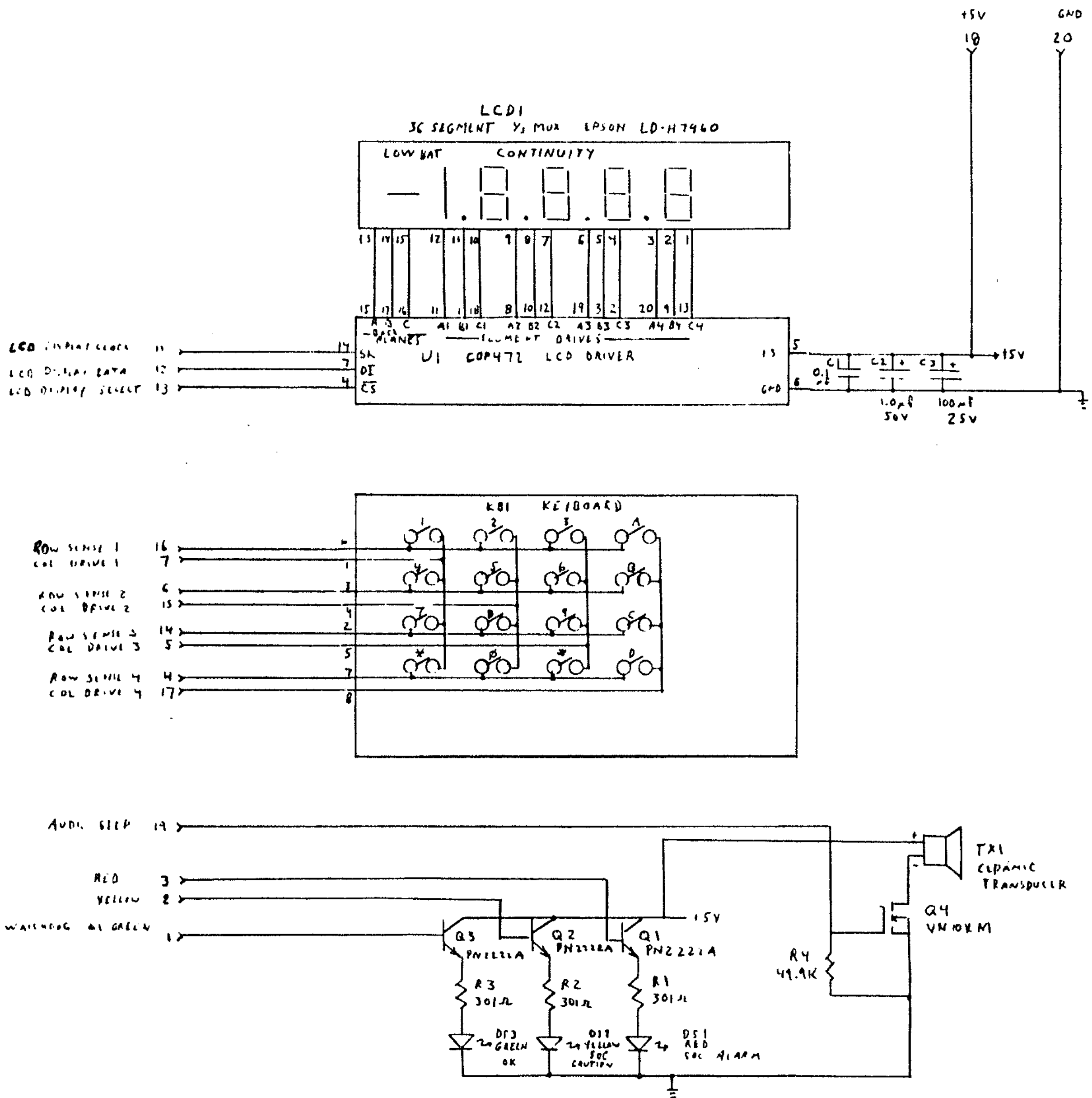


ORIGINAL PAGE IS OF POOR QUALITY

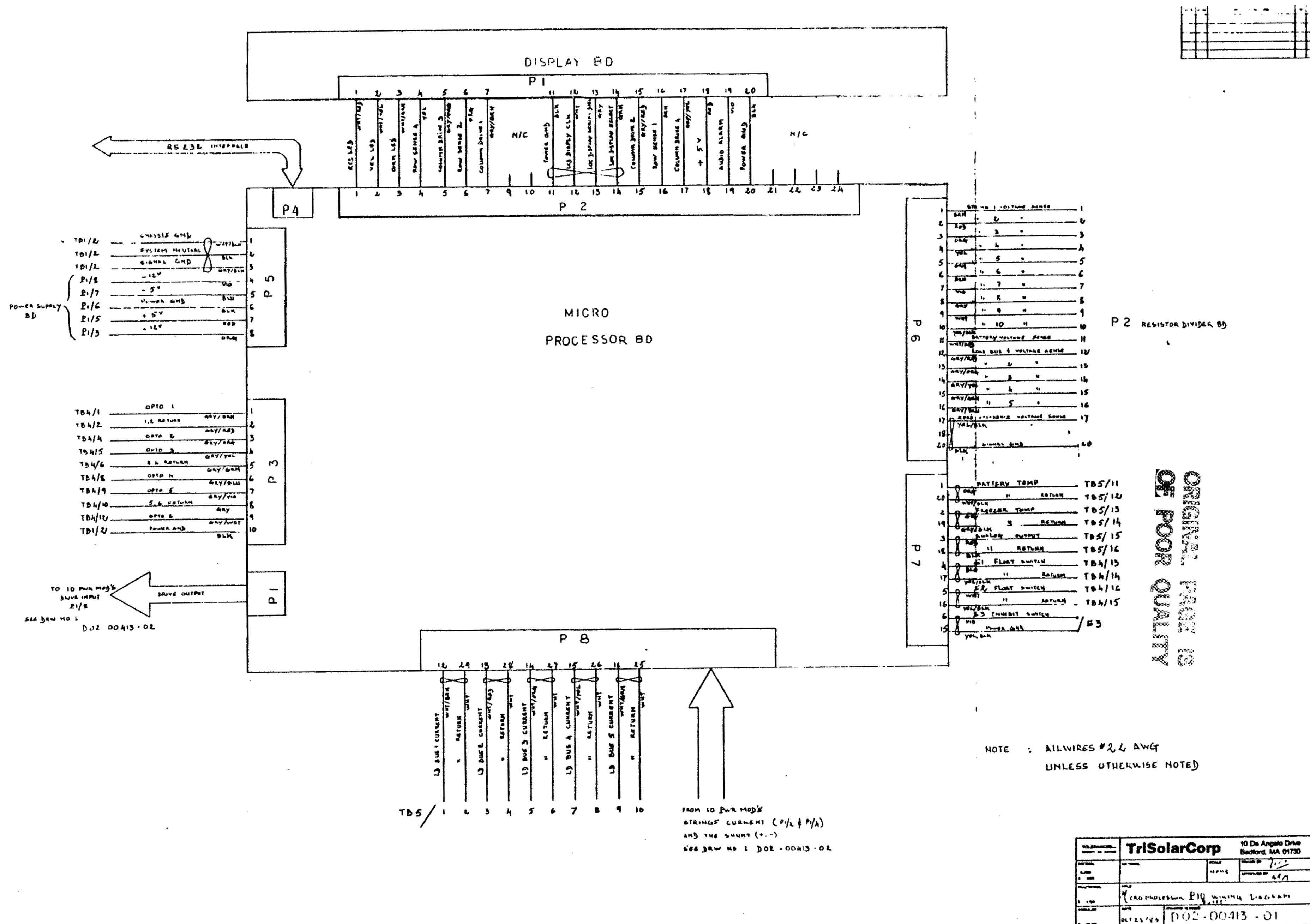
TOLERANCES	TriSolarCorp		10 De Angelo Drive, Bedford, MA 01730
DECIMAL	MATERIAL	SCALE	DRAWN BY ARM
FRACTIONAL	TITLE MICROPROCESSOR CONTROLLER TEST CONFIGURATION		APPROVED BY ARM
ANGULAR	DATE 12-5-83	DRAWING NUMBER C02-00416-00 A	

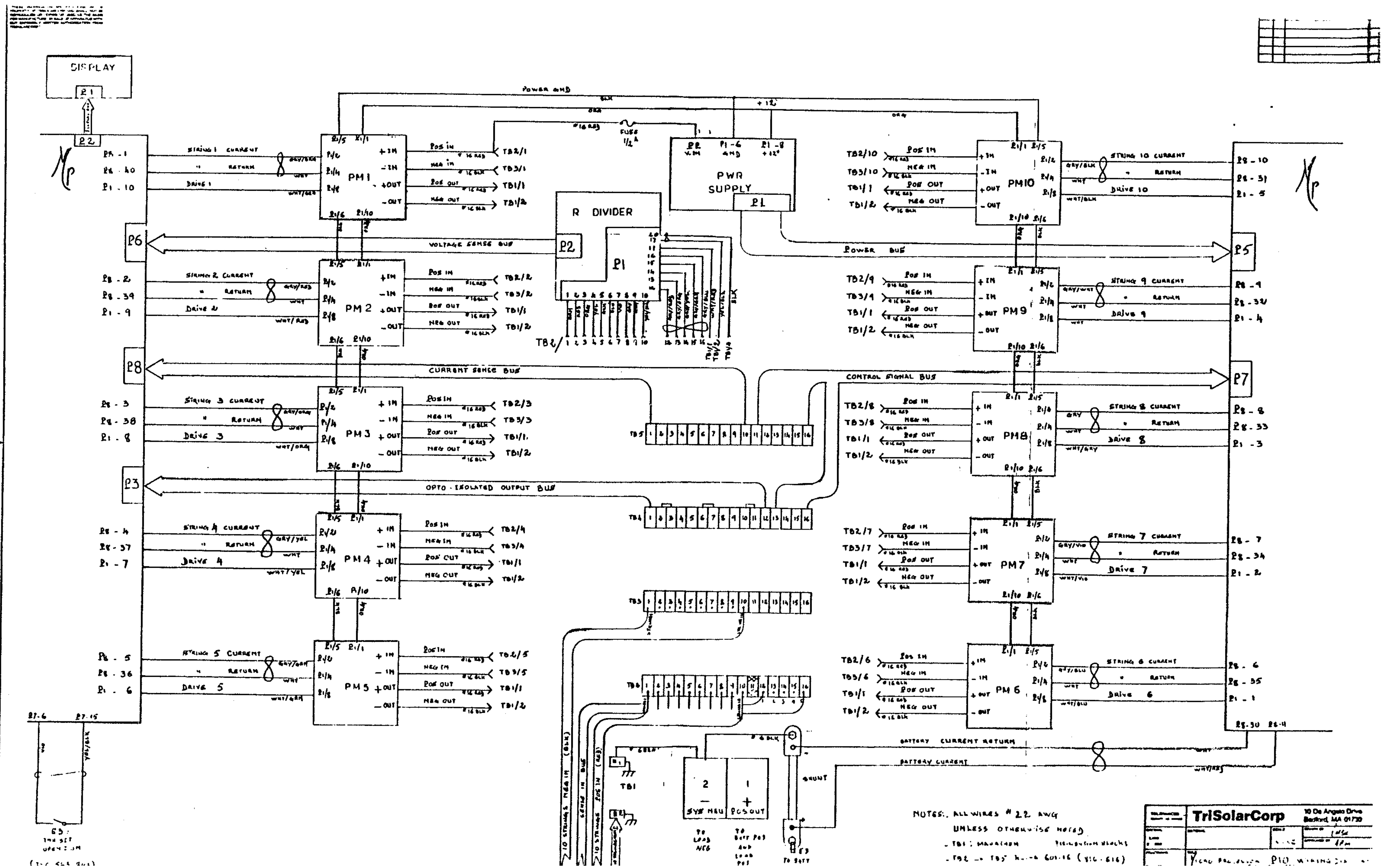
DATE	SYM	REVISION RECORD	DR	CK

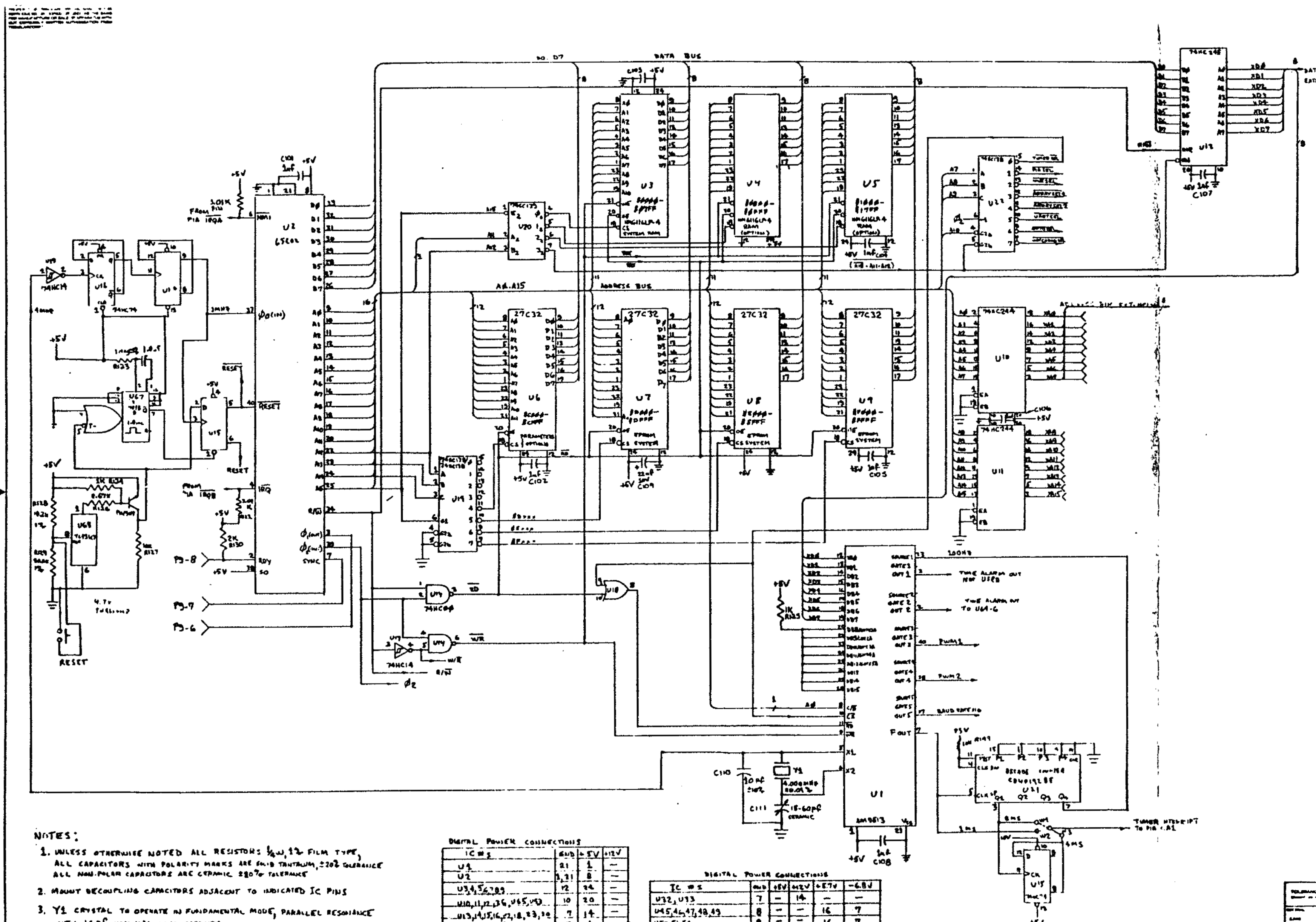
REPLACES C0200404-00



TOLERANCES	TriSolarCorp	10 De Angelo Drive Bedford, MA 01730	
DECIMAL X.XXX ± .005	MATERIAL	SCALE	DRAWN BY ARM APPROVED BY


DISPLAY BD
P1
P2
P4
RS232 INTERFACE
MICRO PROCESSOR BD
P5
P3
P1
P6
P7
P8
N/C
TB1/2 CHASSIS GND
TB1/2 SYSTEM NEUTRAL
TB1/2 SIGNAL GND
POWER SUPPLY BD
P1/8 -12V
P1/7 -5V
P1/6 POWER GND
P1/5 +5V
P1/3 +12V
TB4/1 OPTO 1
TB4/2 1,2 RETURN
TB4/4 OPTO 2
TB4/5 OPTO 3
TB4/6 3,4 RETURN
TB4/8 OPTO 4
TB4/9 OPTO 5
TB4/10 5,6 RETURN
TB4/12 OPTO 6
TB1/2 POWER GND
TO 10 PWR MOD'S DRIVE INPUT P1/8
SEE DRW NO 2 D02 00413-02
DRIVE OUTPUT
P2 RESISTOR DIVIDER BD
BATTERY VOLTAGE SENSE
LOAD BUS 1 VOLTAGE SENSE
BATTERY TEMP
FREEZER TEMP
ANALOG OUTPUT
#1 FLOAT SWITCH
#2 FLOAT SWITCH
#3 INHIBIT SWITCH
POWER GND
TB5/11
TB5/12
TB5/13
TB5/14
TB5/15
TB5/16
TB4/13
TB4/14
TB4/16
TB4/15
TB5
LD BUS 1 CURRENT
LD BUS 2 CURRENT
LD BUS 3 CURRENT
LD BUS 4 CURRENT
LD BUS 5 CURRENT
RETURN
FROM 10 PWR MOD'S STRINGS CURRENT (P1/2 & P1/4) AND THE SHUNT (+,-)
SEE DRW NO 2 D02-00413-02
NOTE: ALL WIRES #22 AWG UNLESS OTHERWISE NOTED
TriSolarCorp
10 De Angelo Drive Bedford, MA 01730
MICROPROCESSOR BD WIRING DIAGRAM
D02-00413-01
ORIGINAL PAGE IS OF POOR QUALITY


DISPLAY
POWER GND
+12
FUSE 1/2 A
PWR SUPPLY
R DIVIDER
PM1
PM2
PM3
PM4
PM5
PM6
PM7
PM8
PM9
PM10
VOLTAGE SENSE BUS
CURRENT SENSE BUS
OPTO - ISOLATED OUTPUT BUS
POWER BUS
CONTROL SIGNAL BUS
BATTERY CURRENT RETURN
BATTERY CURRENT
SHUNT
TB1
SYS NEG
POS OUT
NOTES: ALL WIRES #22 AWG UNLESS OTHERWISE NOTED
TriSolarCorp
10 De Angelo Drive Bedford, MA 01730



ORIGINAL PAGE IS OF POOR QUALITY

NOTES:

1. UNLESS OTHERWISE NOTED ALL RESISTORS ¼W, 1% FILM TYPE, ALL CAPACITORS WITH POLARITY MARKS ARE SOLID TANTALUM, ±20% TOLERANCE ALL NON-POLAR CAPACITORS ARE CERAMIC ±20% TOLERANCE
2. MOUNT DECOUPLING CAPACITORS ADJACENT TO INDICATED IC PINS
3. Y1 CRYSTAL TO OPERATE IN FUNDAMENTAL MODE, PARALLEL RESONANCE WITH 16pF NOMINAL LOAD CAPACITY.

DIGITAL POWER CONNECTIONS

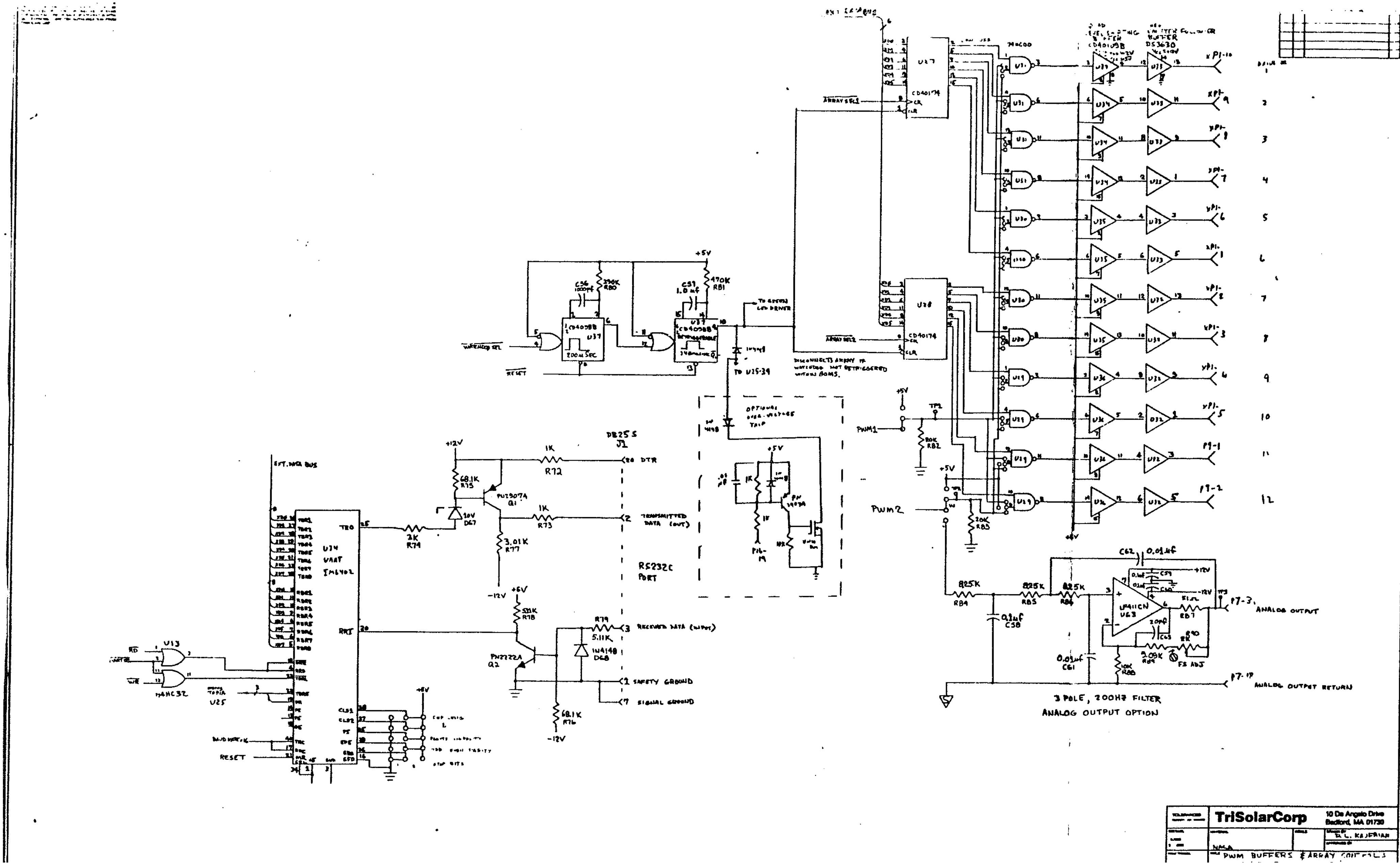
IC #s	GND	+5V	+12V
U1	21	1	—
U2	1,21	8	—
U3,4,5,6,7,8,9	12	24	—
U10,11,12,36,U65,U53	10	20	—
U13,14,15,16,17,18,23,30	7	14	—
U31,U44,U56	7	14	—
U19,20,21,22,23,37,54,67	8	16	—
U25	1	20	—
U24	3	1	—
U34,35,3-	8	1	16

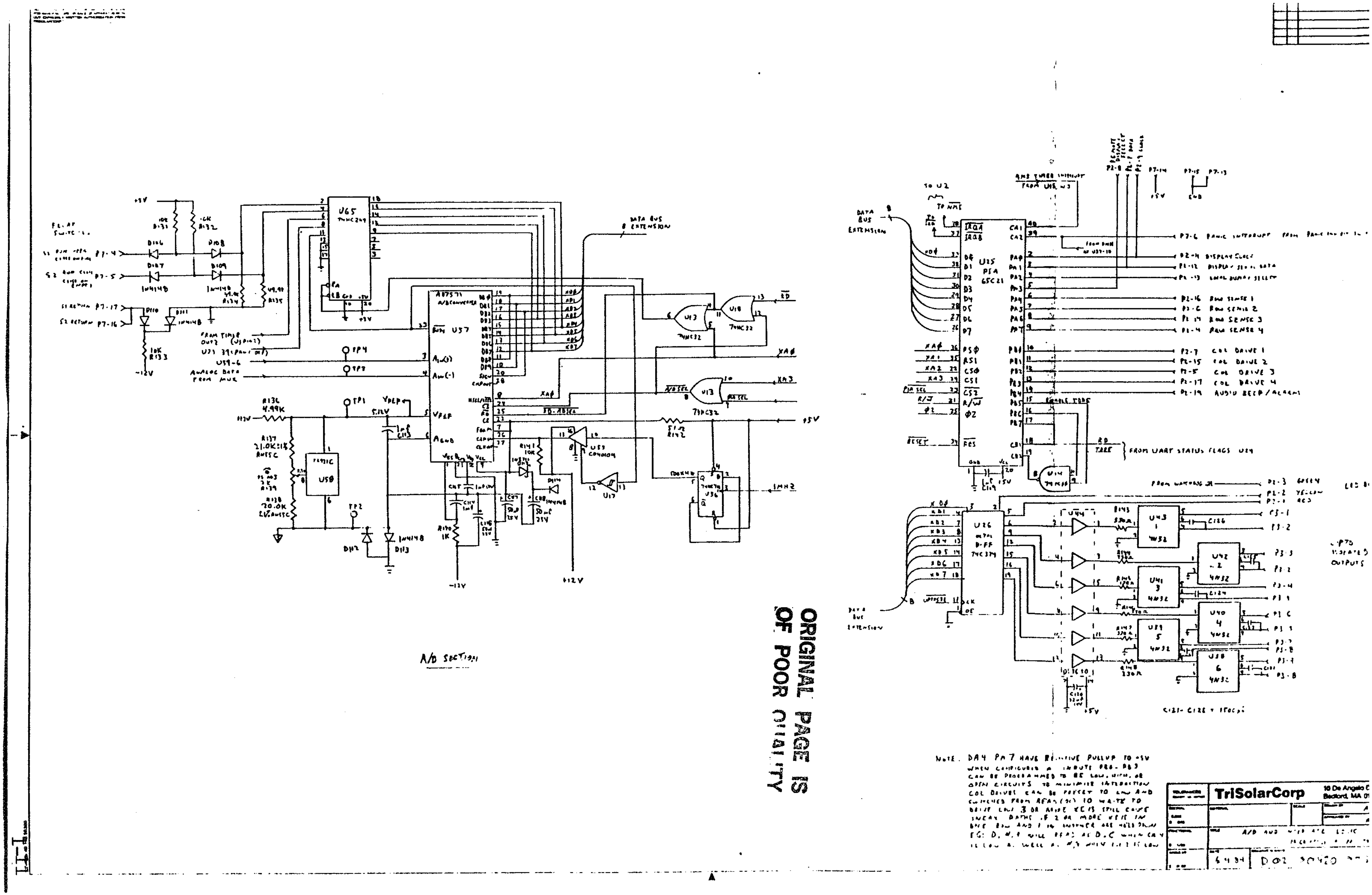
DIGITAL POWER CONNECTIONS

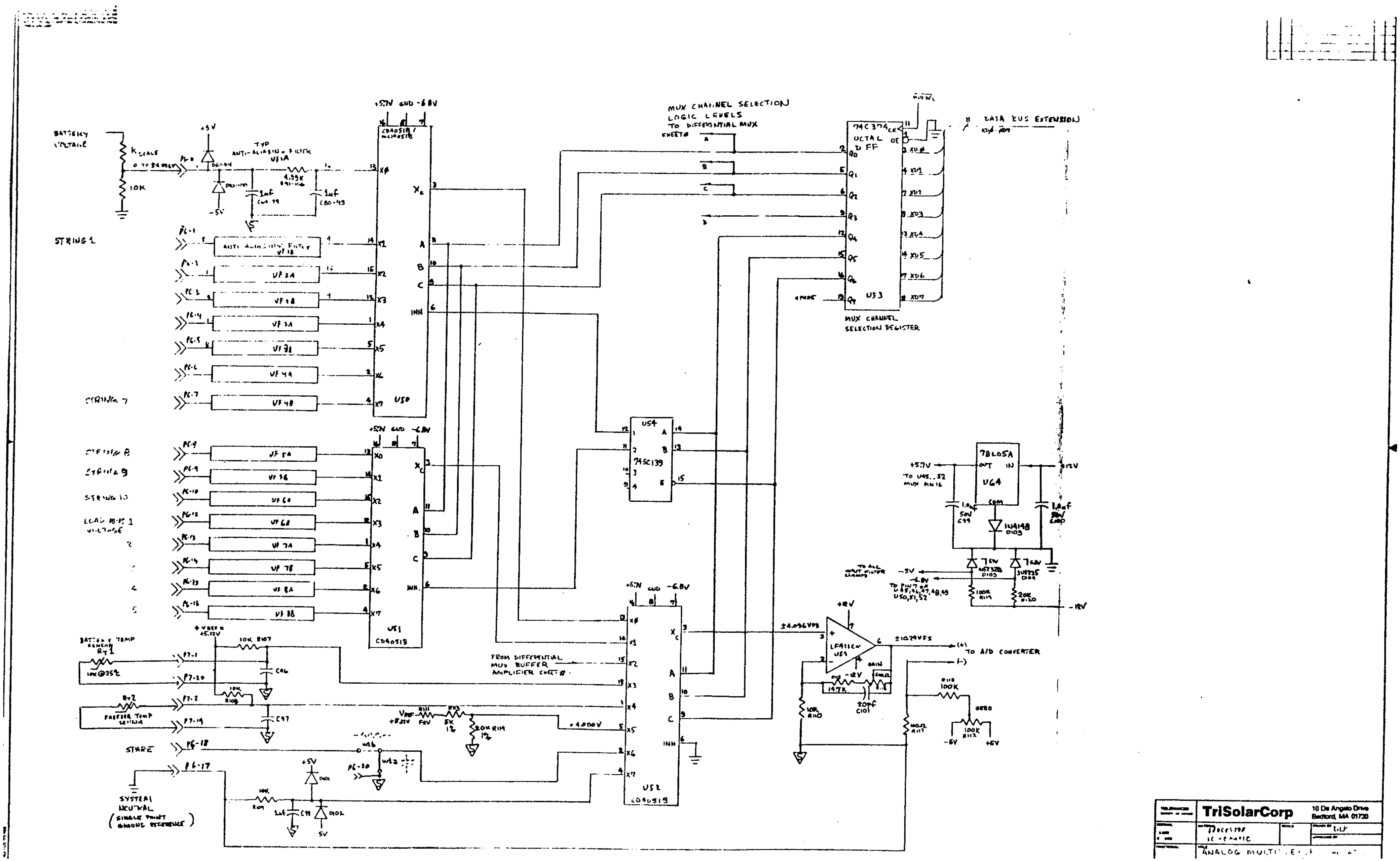
IC #s	GND	+5V	+12V	+5.7V	-6.8V
U32,U33	7	—	14	—	—
U45,46,47,48,49	8	—	—	16	7
U50,51,52	8	—	—	16	7
U57	21	—	—	—	—
U68	6	—	—	—	—
U4-,41,42,43,38,3-	2	—	—	—	—

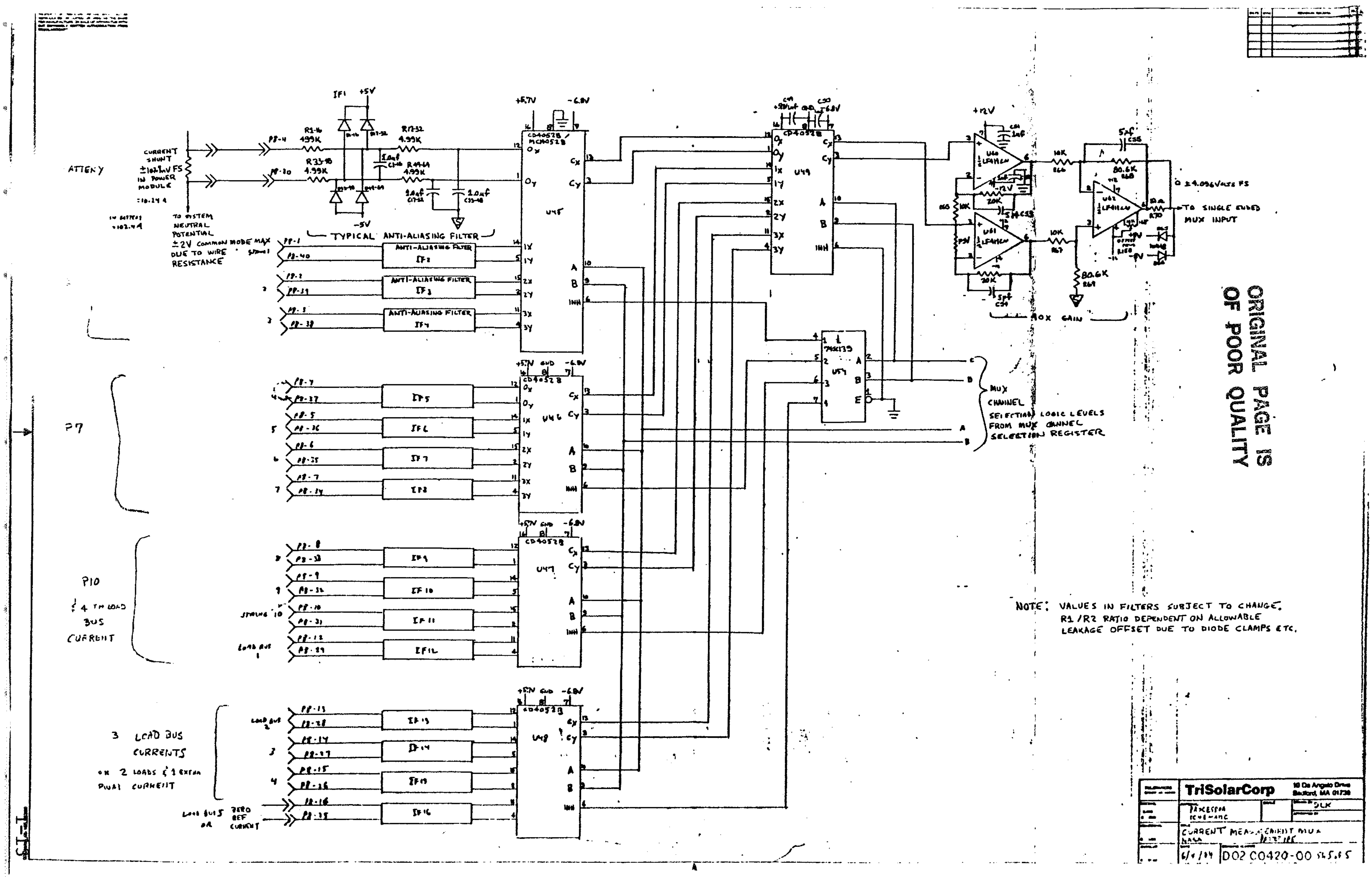
TriSolarCorp
10 De Angelo Dr.
Bedford, MA 017

D02 00420-00


TriSolarCorp
10 De Angelo Drive
Bedford, MA 01730
NASA
PWM BUFFERS & ARRAY CONTROLS
3 POLE, 200HZ FILTER
ANALOG OUTPUT OPTION
RS232C
PORT
DB25S
J1
20 DTR
2 TRANSMITTED DATA (OUT)
3 RECEIVED DATA (INPUT)
2 SAFETY GROUND
7 SIGNAL GROUND
U34
UART
IM6402
PN2907A
Q1
PN2222A
Q2
1N4148
D68
R72
R73
R74
R75
R76
R77
R78
R79
U13
74HC32
U25
RESET
WATCHDOG SEL
CD4098B
U37
200mSEC
U27
CD40174
U28
CD40174
ARRAY SEL1
ARRAY SEL2
PWM1
PWM2
+5V
+12V
-12V
LF411CN
U63
C58
C61
C62
R84
R85
R86
R87
R88
R89
F3 ADJ
ANALOG OUTPUT
ANALOG OUTPUT RETURN


U65
74HC244
AD7571
A/D CONVERTER
U57
DATA BUS
8 EXTENSION
U17
74HC32
U18
U13
U58
A/D SECTION
U25
PIA
65C21
U26
74C374
4N32
FROM UART STATUS FLAGS
TriSolarCorp
10 De Angelo
Bedford, MA


MUX CHANNEL SELECTION
LOGIC LEVELS
TO DIFFERENTIAL MUX
DATA BUS EXTENSION
74C374
OCTAL D FF
U53
MUX CHANNEL
SELECTION REGISTER
BATTERY
VOLTAGE
10K
TYP
ANTI-ALIASING FILTER
STRING 1
STRING 7
STRING 8
STRING 9
STRING 10
CD4051B
U50
U51
U52
U54
74C139
78L05A
U64
1N4148
FROM DIFFERENTIAL
MUX BUFFER
AMPLIFIER
LF411CN
TO A/D CONVERTER
GAIN
ZERO
BATTERY TEMP
SENSOR
FREEZER TEMP
SENSOR
SPARE
SYSTEM
NEUTRAL
(SINGLE POINT
GROUND REFERENCE)
TriSolarCorp
10 De Angelo Drive
Bedford, MA 01730


ATTEN Y
CURRENT SHUNT ±100mV FS IN POWER MODULE
TO SYSTEM NEUTRAL POTENTIAL
±2V COMMON MODE MAX DUE TO WIRE RESISTANCE
TYPICAL ANTI-ALIASING FILTER
ANTI-ALIASING FILTER
R1-16 499K
R35-48 4.99K
R17-32 4.99K
R49-64 4.99K
+5V
-5V
+5.7V
-6.8V
CD4052B / MC14052B
U45
U46
U47
U48
U49
CD4052B
74HC139
U57
U60
U61
U62
LF441CN
MUX CHANNEL SELECTION LOGIC LEVELS FROM MUX CHANNEL SELECTION REGISTER
TO SINGLE ENDED MUX INPUT
0 ±4.096Volts FS
BOX GAIN
80.6K
10K
20K
P7
P10
4 TH LOAD BUS CURRENT
3 LOAD BUS CURRENTS
OR 2 LOADS & 1 EXTRA PWAL CURRENT
ZERO REF CURRENT
NOTE: VALUES IN FILTERS SUBJECT TO CHANGE, R1 /R2 RATIO DEPENDENT ON ALLOWABLE LEAKAGE OFFSET DUE TO DIODE CLAMPS ETC.
ORIGINAL PAGE IS OF POOR QUALITY
TriSolarCorp
10 De Angelo Drive Bedford, MA 01730
CURRENT MEASUREMENT MUX
NASA
D02 C0420-00

APPENDIX II

PSEUDOCODE LISTING

APPENDIX II

SUMMARY OF SOFTWARE DESIGN
MAXIMUM POWER TRACKING PHOTOVOLTAIC SYSTEM CONTROLLER

prepared by Thomas A. Maier
Fellow
Mellon Institute Computer Engineering Center
March 22, 1984

CONTENTS

1. The Use of Pseudocode

The majority of the information regarding the software package developed by the Mellon Institute Computer Engineering Center (MI/CEC) for the captioned TriSolar project is contained in the pseudocode listings which accompany this summary. The pseudocode is a fictitious high level language used to document and describe the assembly code used in each of the routines. The use of flowcharts has, in the past, been cumbersome to keep updated as corrections and modifications are incorporated into a piece of code. A pseudocode listing, on the other hand, is easily understood and can be incorporated as part of the comments in the assembly language code. The size of the source code file in this particular project, however, required that the pseudocode be kept in a separate file.

The syntax of the pseudocode generally follows the PL/M-86 language which was selected because it was more "english-like" than many of the other programming languages. It should be pointed out that occasionally constructs not supported by PL/M-86 were required. When this happened, the syntax rules were "bent" somewhat to allow english phrases or carefully selected syntax from other languages to fill the void. The meaning is still clear to anyone with familiarity with programming in general.

By first writing the routines in the pseudocode and then translating the resulting pseudocode, line by line, very clean, structured assembly code can be generated. If this technique was carefully adhered to, routines containing hundreds of lines of code can be generated with few, if any, errors.

2. General Comments

The remainder of this summary details some of the other information that is not reflected in the pseudocode listings, such as; information about the A/D routine, A/D channel allocations, Signal Averaging and similar topics.

2.1. Analog to Digital Converter Interrupt Routine

As a result of a negative transition on the 4 millisecond NMI line, the A/D interrupt handler gathers 13 channels of data, one at a time, and stores the results in memory. To accomplish this, the routine sets a mux channel address, waits for settling to occur, starts the conversion, polls for conversion complete, formats and stores the result and then repeats the process for each successive channel. In order to conserve time, the formating, which amounts to rearranging the word bit pattern and changing from sign magnitude to 2's completement notation, is interleaved with the 80 microsecond conversion time of the next channel.

The data read from the A/D is returned in 2 bytes, shown here msbyte first:

b7 b6 b5 b4 b3 sign b9 b8 : bsy t1 sw2 sw1 w b2 b1 b(

where bX = data bit X
sign = sign bit, "1" = negative
bsy = A/D busy line, "1" = done
t1 = 30 minute alarm signal
w = watchdog/panic interrupt line
sw1 = inhibit switch,
sw2 = motor thermostat

which must be rearranged to the form:

sign sign sign sign sign sign b9 b8 : b7 b6 b5 b4 b3 b2 b1 b0

Notice the extended sign bit and the elimination of th miscellaneous input bits. It is important to retain the other dat bits, which are used elsewhere, because random reads to the A/I (without a corresponding write, to initiate a conversion cycle) plac the A/D into an indeterminate state. Since these data bits ar available during each read of the A/D, they are retained when the A/I is read during the seldom read channel and are stored in alarm_30 fo use at a later time.

2.1.1. A/D Channel Allocations

Channels are broken up into two groups, the "often-read's" and the "seldom-read's". As implemented in version 1.X of the software, the first group contains 12 channels (i.e. 10 branch currents and the battery current and voltage) that are read every time the routine is called. The latter group contains all of the other channels and are read in a "round robin" fashion, one per pass thru the routine. The following is a list of the "often_read" and "seldom_read" channels showing the mux addresses and channel number associated with each.

OFTEN READ CHANNELS

signal name	mux address	often-read channel number
battery volts	$00	0
battery current	$20	1
branch #1 current	$21	2
branch #2 current	$22	3
branch #3 current	$23	4
branch #4 current	$24	5
branch #5 current	$25	6
branch #6 current	$26	7

branch #7 current	$27	8
branch #8 current	$28	9
branch #9 current	$29	A
branch #10 current	$2A	B

SELDOM READ CHANNELS

signal name	mux address	seldom-read channel number
branch #1 volts	$01	0
branch #2 volts	$02	1
branch #3 volts	$03	2
branch #4 volts	$04	3
branch #5 volts	$05	4
branch #6 volts	$06	5
branch #7 volts	$07	6
branch #8 volts	$10	7
branch #9 volts	$11	8
branch #10 volts	$12	9
load bus # 1 volts	$13	A
load bus # 2 volts	$14	B
load bus # 3 volts	$15	C
load bus # 4 volts	$16	D
load bus # 5 volts	$17	E
load bus #1 current	$2B	F
load bus #2 current	$2C	10
load bus #3 current	$2D	11
load bus #4 current	$2E	12
load bus #5 current	$2F	13
battery temp	$30	14
freezer temp	$40	15
80% of Vref	$50	16
zero voltage reference	$70	18

2.1.2. Signal Averaging

The raw, often_read channel data is stored (2's complement form) in a 12 x 16 WORD array based at often_read_base (at location $0320). The array is arranged in the following manner. The table shows the address at which the lsbyte of each reading for each channel of the "often_read's" is stored.

Signal Averaging Array

channel numbers--> reading number	0	1	2	3	4	5	6	7	8	9	A	
1	320	340	360	380	3A0	3C0	3E0	400	420	440	460	4
2	322	342	362	382	3A2	3C2	3E2	402	422	442	462	4
3	324	344	364	384	3A4	3C4	3E4	404	424	444	464	4
4	326	346	366	386	3A6	3C6	3E6	406	426	446	466	4
5	328	348	368	388	3A8	3C8	3E8	408	428	448	468	4
6	32A	34A	36A	38A	3AA	3CA	3EA	40A	42A	44A	46A	4
7	32C	34C	36C	38C	3AC	3CC	3EC	40C	42C	44C	46C	4
8	32E	34E	36E	38E	3AE	3CE	3EE	40E	42E	44E	46E	4
9	330	350	370	390	3B0	3D0	3F0	410	430	450	470	4
10	332	352	372	392	3B2	3D2	3F2	412	432	452	472	4
11	334	354	374	394	3B4	3D4	3F4	414	434	454	474	4
12	336	356	376	396	3B6	3D6	3F6	416	436	456	476	4
13	338	358	378	398	3B8	3D8	3F8	418	438	458	478	4
14	33A	35A	37A	39A	3BA	3DA	3FA	41A	43A	45A	47A	4
15	33C	35C	37C	39C	3BC	3DC	3FC	41C	43C	45C	47C	4
16	33E	35E	37E	39E	3BE	3DE	3FE	41E	43E	45E	47E	4

Note often_read_base = $320

When 16 sets of data have been gathered the dataset_ready_flag i set. This flag is used as the handshake semaphore. While th dataset_ready_flag = 1, the background program has not, as yet, converted and used the last data set. When the dataset_ready_flag i found to be asserted, the signal_av routine, which runs in th background, sums the 16 individual readings of each channel (i.e. on of the above columns) and divides the result by 16 to produce a average for that channel. This result is placed into the proper location in the dump_state array.

The calculation of total_chgr_I, i.e. the total array current is done in this routine as well. Each of the branch currents is summed and the total is stored in the dump_state array as outlined below. Since the A/D reading of any given branch is a maximum of 10 bits, the sum of the 10 strings can never exceed the 16 bit word reserved for it.

All of the seldom_read channels are loaded directly into dump_array as they are read without signal averaging.

The following figure shows the arrangement of the data in the "dump_state" array.

2.1.3. A/D Channel Storage Locations

Often Read Channels:

addr	variable	description
0008	battery_V:	averaged battery voltage
000A	battery_I:	averaged battery current
000C	branch1:	averaged branch 1 current
000E	branch2:	averaged branch 2 current
0010	branch3:	averaged branch 3 current
0012	branch4:	averaged branch 4 current
0014	branch5:	averaged branch 5 current
0016	branch6:	averaged branch 6 current
0018	branch7:	averaged branch 7 current
001A	branch8:	averaged branch 8 current
001C	branch9:	averaged branch 9 current
001E	branch10:	averaged branch 10 current

Seldom Read Channels:

0020	br_volts1:	branch 1 voltage
0022	br_volts2:	branch 2 voltage
0024	br_volts3:	branch 3 voltage
0026	br_volts4:	branch 4 voltage
0028	br_volts5:	branch 5 voltage
002A	br_volts6:	branch 6 voltage
002C	br_volts7:	branch 7 voltage
002E	br_volts8:	branch 8 voltage
0030	br_volts9:	branch 9 voltage
0032	br_volts10:	branch 10 voltage
0034	bus_volts1:	load bus #1 voltage
0036	bus_volts2:	load bus #2 voltage
0038	bus_volts3:	load bus #3 voltage
003A	bus_volts4:	load bus #4 voltage
003C	bus_volts5:	load bus #5 voltage
003E	bus_amps1:	load bus #1 current
0040	bus_amps2:	load bus #2 current
0042	bus_amps3:	load bus #3 current
0044	bus_amps4:	load bus #4 current
0046	bus_amps5:	load bus #5 current
0048	bat_temp:	battery temperature
004A	frez_temp:	freezer temperature
004C	V_ref:	reading of .8 Vref
004E	zero_ref:	zero voltage reference
0050	total_chgr_I:	current total charger current

Since the 4 millsecond interrupt is the only means for providing system timing, the one_second and 80_msec timers and flags are both controlled by this routine. Both timers (located in memory) are decremented each pass and reloaded when they reach 0. When either

C - 2

timer "times out", the respective flag is set to indicate to th background routine that the time period has ellapsed.

Settle_time is set by the max_pwr_track routine every time that change is made to the PWM counters. If settle_time is non-zero, all o the data gathering code in this routine is skirted. This gives th power module time to settle before data is again taken.

This routine also invokes scan_kybd on each pass to handle th scanning of the hex keypad on the front panel.

2.2. Operator Interface

The following four sections outline the functions that ar available to the user either via the hexadecimal keypad on the fron panel or the serial port (300 baud only). The channel numbers use for accessing the different quantities are as follows: (Note th difference between the user channel number and the raw channel number, which is used only with the Q command through the serial port.)

OFTEN READ CHANNELS

signal name	user ch number (serial)	user ch number (keypad)	display pattern	raw ch number
battery volts	E00	A00	xxx.x	Q00
battery current	I00	B00	xxx.x	Q01
branch #1 current	I01	B01	xx.xx	Q02
branch #2 current	I02	B02	xx.xx	Q03
branch #3 current	I03	B03	xx.xx	Q04
branch #4 current	I04	B04	xx.xx	Q05
branch #5 current	I05	B05	xx.xx	Q06
branch #6 current	I06	B06	xx.xx	Q07
branch #7 current	I07	B07	xx.xx	Q08
branch #8 current	I08	B08	xx.xx	Q09
branch #9 current	I09	B09	xx.xx	Q10
branch #10 current	I10	B10	xx.xx	Q11

SELDOM READ CHANNELS

signal name	user ch number (serial)	user ch number (keypad)	display pattern	raw ch number
branch #1 volts	E01	A01	xxx.x	Q12
branch #2 volts	E02	A02	xxx.x	Q13
branch #3 volts	E03	A03	xxx.x	Q14
branch #4 volts	E04	A04	xxx.x	Q15
branch #5 volts	E05	A05	xxx.x	Q16
branch #6 volts	E06	A06	xxx.x	Q17

branch #7 volts	E07	A07	xxx.x	Q18
branch #8 volts	E08	A08	xxx.x	Q19
branc9 #9 volts	E09	A09	xxx.x	Q20
branch #10 volts	E01	A10	xxx.x	Q21
load bus #1 volts	E31	A31	xxx.x	Q22
lö̈ad bus #2 volts	E32	A32	xxx.x	Q23
load bus #3 volts	E33	A33	xxx.x	Q24
load bus #4 volts	E34	A34	xxx.x	Q25
load bus #5 volts	E35	A35	xxx.x	Q26
load bus #1 current	I31	B31	xxx.x	Q27
load bus #2 current	I32	B32	xxx.x	Q28
load bus #3 current	I33	B33	xxx.x	Q29
load bus #4 current	I34	B34	xxx.x	Q30
load bus #5 current	I35	B35	xxx.x	Q31
battery temp	D36	D36	xxx.x	Q32
freezer temp	D37	D37	xxx.x	Q33
80% of Vref	E38	A38	x.xxx	Q34
zero voltage reference	E39	A39	x.xxx	Q35
total charger current	I40	B40	xxx.x	Q37
state of charge	D00	D00	xxx	-
corrected state of chg	D40	D40	xxx	-
equal_count	D41	D41	xxx	-
pwm_value	D42	D42	xxx	-

The following are calculated at display time:

signal name	user ch number (serial)	user ch number (keypad)	display pattern
battery power	P00	C00	xxxxx
branch #1 power	P01	C01	xxxxx
branch #2 power	P02	C02	xxxxx
branch #3 power	P03	C03	xxxxx
branch #4 power	P04	C04	xxxxx
branch #5 power	P05	C05	xxxxx
branch #6 power	P06	C06	xxxxx
branch #7 power	P07	C07	xxxxx
branch #8 power	P08	C08	xxxxx
branch #9 power	P09	C09	xxxxx
branch #10 power	P10	C10	xxxxx

2.2.1. Hexadecimal Keypad Functions

Some of the following functions, i.e. those noted with (mult), are "multimeter functions". This means that when this function is selected, every 320 msec the quality is measured, converted as necessary and redisplayed.

Definition of notation:

a - any of the alphanumeric keys (i.e. not "*" or "#")

nn - channel/device specification consisting of any two numeric keys

mm or mmm - data specification consisting of any two or three numeric

NOTE: all inputs are assumed to be "fixed format" i.e., if the specif calls for 3 digits, leading zeroes must be added to make the input 3 long.

2.2.1.1. Public Functions

sequence	function
*	clear function
aaaa#	activate password-accessible functions if user password matches(see below)
Ann#	read channel nn voltage (mult)
Bnn#	read channel nn current (mult)
Cnn#	read channel nn power (mult)
Dnn#	read misc data channels (mult)
AA#	display software version number
BB#	read time, hours and minutes (mult)
CC#	initiate "dump" of machine state to serial port
DD#	not used

2.2.1.2. Password-Accessible Functions

In order to access these functions, the user must have successfully entered the 4 character password

sequence function

===:

Anmmm#	set load shed threshold for load n at mmm
Bnmmm#	set load restore threshold for load n at mmm
Cmmm#	set the initial percentage SOC at mmm%
Dnnm#	set device nn to condition m, where m must be either a "1" (ON) or a "0" (OFF)

device 0	audible alarm
device 1-6	user load requests 1 thru 6
device 7-11	overload trip resets for loads 1 thru 5
device 12-17	PWM buffer #1 controls 1 thru 6
device 18-23	PWM buffer #2 controls 1 thru 6
device 24-25	Yellow, and Red LEDs

AA#	initiate lamp and annunciator test (accessable only in test/cal mode)
BB#	toggle from run to test/cal mode (system comes up in the run mode)
CChhmm#	set time, hours and minutes
DD#	cancel password authorization

2.2.1.3. Debug Monitor Functions

These functions are accessed via the serial port. They are listed in upper and lower case letters for comparison with the commands above. In order to permit maximum flexibility with respect to terminals, either case is useable in practice.

sequence	function
^H, backspace, del	Deletes last character entered. Echoes backspace, space, backspace to allow overwriting the last character entered when a CRT terminal is used.
^U	Causes CPU to ignore present command line
^Z	Return to command mode
M<addr>	Opens a memory location at the specified address (requires 4 hexadecimal digits). Successive " " (space) characters increment thru memory, while "-" characters decrement thru memory. At any time the contents of

a location may be altered by entering the new data followed by a carriage return.

G<addr>

Begins execution at the specified address. no address is specified, execution begins the present PC location

B<addr>

Places a breakpoint at the specified addre This trace mode will only work on code located in RAM.

X

Removes existing breakpoint.

L<addr>

This permits a program to be downloaded fr a host machine to memory starting at the specified address.

F<start_addr end_addr datum>

Fills the specified memory range with spec byte of data.

.

.

2.2.1.4. Maintenance and Logging Functions

Many of the following functions are essentially identical t those invoked from the hexadecimal keypad, the exceptions being thos commands requiring password authorization. The main difference i that the system response is returned via the serial port. This coul be used for data logging by having an external device request th desired data.

sequence	function
W<pwm timer no.><duty cycle>	Set the Power Module duty cycle to specified value. If duty cycle > max_pwm, the default duty cycle is set to max_pwm
O<digit no.><value>	Display the value in the specified digit on t LCD display.
Enn	Read channel nn voltage
Inn	Read channel nn current
Pnn	Read channel nn power
Dnn	Read misc data channels
Qnn	Query channel nn for "raw" A/D data

N	Display software version number
T	Read time, hours and minutes
Snmmm	Set load shed threshold for load n at mmm
Rnmmm	Set load restore threshold for load n at mmm
Ymmm	Set the initial percentage SOC at mmm%
Jnnm	Set device nn to condition m, where m must be either a "1" (ON) or a "0" (OFF)

device 0	audible alarm
device 1-6	user load requests 1 thru 6
device 7-11	overload trip resets for loads 1 thru 5
device 12-17	PWM buffer #1 controls 1 thru 6
device 18-23	PWM buffer #2 controls 1 thru 6
device 24-25	Yellow, and Red LEDs

A	Initiate lamp and annunciator test (accessable only in test/cal mode)
Z	Toggle between run and test/cal mode, note the system comes up in the test/cal mode
Chhmm	Set time, hours and minutes
K	Initiate "dump" of machine state

2.3. Calculating And Interpreting System Parameters

In order to calculate the hexadecimal values used for system parameters, it is first necessary to classify the type, because each type of parameter must be calculated somewhat differently.

Currents are broken into two types. Battery and load currents have a maximum value of 102.3 amps while the branch currents have a maximum of 10.23 amps. Hence, full scale, i.e. $3FF has two different meanings depending on which current you are referring to. In the first case the reading corresponds to the number of "1/10's" of amps being measured, while in the second case to the number of "1/100's" of amps. Once the desired current has been expressed in the proper units, converting the decimal number to hexadecimal results in the correct parameter value. For example:

```
1. battery_I = 82 amps
             = 820 "1/10's of an amp"
             = (3 * 256) + (3 * 16) + (4 * 1)
             = $0334
```

```
2. branch1 = 7.2 amps
     = 720 "1/100's of an amp"
           = (2 * 256) + (13 * 16)
           = $02D0
```

Calculating voltages is somewhat simpler because, with only 3 exceptions, all voltages are based on a full range value of 409.2 volts. As with the current measurements, most voltage readings represents the number of "1/10's of a volt". However, in the case of the thermister signals (frez_temp and bat_temp) and V_ref the raw A/D count represents the number of "1/1000's of a volt". A full scale reading of $3FF corresponds to 1023 (base 10). If this number is shifted left 2 places, $3FF becomes $FFC which corresponds to 4092. Hence, to convert from decimal volts to hexadecimal volts, divide the voltage by 4 and convert the result to hexadecimal. For example:

```
1. V_ref = 4.092 volts
         = 4092 "1/1000's of a volt"
  4092/4 = 1023
         = (3 * 256) + (15 * 16) + (15 * 1)
         = $03FF

2. abs_min_bat_volt = 95 volts
                    = 950 "1/10's of a volt"
              950/4 = 237
                    = (14 * 16) + (13 * 1)
                    = $00ED
```

Binary percentages for state_of_chg, equal_count and similar variables are represented by a 8 bit number with the binary point located 1 bit in from the MSB, as shown here.

```
           MSB
            |
            X.XXXXXXX
             |
        binary point
```

To calculate binary percentages, use this relation:

binary % = ((decimal % * 128)/100) expressed in hexadecimal

For example:

```
1. maximum binary % = (100% * 128)/100
                    = 128
                    = (8 * 16)
                    = $80

2. shed_th = (20% * 128)/100
```

```
= 26
= (1 * 16) + (10 * 1)
= $1A
```

To calculate the constant used for state_of_chg use the following relation:

```
iscal_bat_cap = 5.49 * 10^10 * iscal * 1/bat_cap
```

For example, if iscal = 1/45000 and bat_cap = 40 A-H then,

```
iscal_bat_cap = 5.49 * 10^10 * (1/(45000 * 40))
              = 30470
```

3. Appendix #1 - Generating System Lookup Tables

Appendix 1 contains copies of the programs, written in the "C" programming language, for the generation of tables to speed the calculation of those variables involving corrections for battery temperature. "C" is commonly used in academic circles and is becoming quite popular in industry. It was used for generation of the values since it is the normal language used here at MI/CEC. The algorithms are quite straight-forward and can be easily translated to BASIC or any other language of choice. For the details of the lookup technique, refer to the Pseudocode listings for the appropriate tables. The first program generates tables for float voltage, equalization voltage and minimum battery voltage, while the second generates the table of values for use with the corrected state of charge routine. Notice that the program outputs each of the tables in two different formats; human readable and machine readable. For examples of the output, see the pseudocode listings for calc_sys_volts and correct_stat_of_chg.

APPENDIX #1

PROGRAM #1 - Generates tables for float_V, equal_V and min_bat_V.

```
#include <math.h>
#include <stdio.h>
main()
{
int     j, i, v, t;
float   x;
double  value, temp;

/* NOTICE!!! THE CONSTANT 2.5 FOUND IN THE EQUATIONS BELOW COMES FROM
   THE FACT THAT THE VOLTAGES ARE STORED IN A FORM THAT IS EQUAL TO
   THE NUMBER OF TENTHS OF VOLTS.  FOR DISPLAY THIS VALUE IS MULTIPLI
   BY 4, SINCE $3FF --> 1023 --> 409.2 VOLTS.

        so, stored value = (input volts * 10)/4
        or, stored value = (input volts * 2.5)          */

        printf("\n\t\tFLOAT VOLTAGE TABLE\n\n");
        for(i=0xD;i <= 66; i=i+1)
                {
                temp = 25-.02092*((64*i)-2560);
                value=2.4 * 54 * 2.5 * (1 + (.0022 * (25-temp)));
                t=i*64;
                v=value;
                value=value * .4;
                printf("%4.0f deg C\t%02x\t%03x\t%04x\t%4.1f\n",temp,
                                i-0xD,t,v,value);
                }
        printf("\n");
        for(i=0xD;i <= 66; i=i+8)
                {
                printf(".word\t");
                for(j=0;j <= 7;j=j+1)
                        {
                        temp = 25-.02092*((64*(j+i))-2560);
                        value=2.4 * 54 * 2.5 * (1 + (.0022 * (25-temp)
                        v=value;
                        printf("$%04x, ",v);
                        }
                printf("\n");
                }
        printf("\14");
        printf("\n\t\tEQUALIZATION VOLTAGE TABLE\n\n");
        for(i=0xD;i <= 66; i=i+1)
                {
                temp = 25-.02092*((64*i)-2560);
                value=2.5 * 54 * 2.5 * (1 + (.0022 * (25-temp)));
                t=i*64;
                v=value;
                value=value * .4;
                printf("%4.0f deg C\t%02x\t%03x\t%04x\t%4.1f\n",temp,
                                i-0xD,t,v,value);
                }
        printf("\n");
```

```
        for(i=0xD;i <= 66; i=i+8)
                {
                printf(".word\t");
                for(j=0;j <= 7;j=j+1)
                        {
                        temp = 25-.02092*((64*(j+i))-2560);
                        value=2.5 * 54 * 2.5 * (1 + (.0022 * (25-temp)));
                        v=value;
                        printf("$%04x, ",v);
                        }
                printf("\n");
                }
        printf("\14");
        printf("\n\t\tMINIMUM BATTERY VOLTAGE TABLE\n\n");
        for(i=0xD;i <= 66; i=i+1)
                {
                temp = 25-.02092*((64*i)-2560);
                value=1.9 * 54 * 2.5 * (1 + (.0022 * (25-temp)));
                t=i*64;
                v=value;
                value=value * .4;
                printf("%4.0f deg C\t%02x\t%03x\t%04x\t%4.1f\n",temp,
                                        i-0xD,t,v,value);
                }
        printf("\n");
        for(i=0xD;i <= 66; i=i+8)
                {
                printf(".word\t");
                for(j=0;j <= 7;j=j+1)
                        {
                        temp = 25-.02092*((64*(j+i))-2560);
                        value=1.9 * 54 * 2.5 * (1 + (.0022 * (25-temp)));
                        v=value;
                        printf("$%04x, ",v);
                        }
                printf("\n");
                }
}
```

ORIGINAL PAGE IS OF POOR QUALITY

PROGRAM #2 - Generates table for correct_state_of_chg.

```
#include <math.h>
#include <stdio.h>
main()
{
int     j, i, v, t;
float   x, coeff;
double  value, temp;

        printf("\n\tCORRECT STATE OF CHARGE TABLE\n\n");
        for(i=0xD;i <= 66; i=i+1)
                {
                temp = 25-.02092*((64*i)-2560);
```

```
                if (temp > 25)
                        coeff = .0022;
                else
                        coeff = .0075;
                value=256 * (1 + (coeff * (temp-25)));
                t=i*64;
                v=value;
                value=value/2.56;
                printf("%4.0f deg C\t%02x\t%03x\t%04x\t%4.0f%%\n",te
                                                i-0xD,t,v,value);
                }
        printf("\n");
        for(i=0xD;i <= 66; i=i+8)
                {
                printf(".word\t");
                for(j=0;j <= 7;j=j+1)
                        {
                        temp = 25-.02092*((64*(j+i))-2560);
                        if (temp > 25)
                                coeff = .0022;
                        else
                                coeff = .0075;
                        value=256 * (1 + (coeff * (temp-25)));
                        t=i*64;
                        v=value;
                        printf("$%04x, ",v);
                        }
                printf("\n");
                }
}
%
```

4. Appendix #2 - Pseudocode Listings

The following is a table of contents for the pseudocode listings. Note that the order of modules is the same as in the machine code PROMs.

TABLE OF CONTENTS FOR PSEUDOCODE LISTINGS

```
XXXXX           XXXXX   XXXXX    XXXX   X    X
X    X          X    X  X    X  X    X  XX  XX
X    X          X    X  X    X  X    X  X XX X
X    X          XXXXX   XXXXX   X    X  X    X
X    X          X       X   X   X    X  X    X
XXXXX           X       X    X   XXXX   X    X
```

```
putchar:
        disable system interrupts;
        call putchar_wo(accum);
        enable system interrupts;
    end putchar;

;*************************************************************************
; This routine places the string defined by the starting address in string_p
; and place it, character by character into the output buffer by calling
; the putchar routine that does not re-enable system interrupts, putchar_wo.
; This routine is used in the serial port handler.
;*************************************************************************

Pseudocode:

msg_hndlr_wo:
        save regs;
        i = 0;
        do while (string_ptr[i] <> 0)
            call putchar_wo(string_ptr[i];
            i = i + 1;
        end;
        unsave regs;
    end msg_hndlr_wo;

;*************************************************************************
; This routine is identical to the above, except that it uses putchar instea
; of putchar_wo and is to be used under normal string output circumstances.
;*************************************************************************

Pseudocode:

msg_hndlr:
        save regs;
        i = 0;
        do while (string_ptr[i] <> 0)
            call putchar(string_ptr[i];
            i = i + 1;
        end;
        unsave regs;
    end msg_hndlr;

;*************************************************************************
; This routine retrieves the next two ascii characters in the command_buffer
; converts them into a single hexadecimal value and return it via the
; accumulator.  Should either of the characters be something other than
; 0-9, a-f or A-F; the ascii_hex_flag is cleared.
;*************************************************************************

Pseudocode:

getbyte:
        save regs;
        accum = command_buffer[cmd_in_ptr];
        cmd_in_ptr = cmd_in_ptr + 1;
```

```
        call a_to_h(accum);
        if (accum <> $FF)
            then do;
                temp = accum << 4;
                accum = command_buffer[cmd_in_ptr];
                cmd_in_ptr = cmd_in_ptr + 1;
                call a_to_h(accum);
                if (accum <> $FF)
                    then do;
                        accum = accum + temp;
                        ascii_hex_flag = 1;
                    end;
                    else ascii_hex_flag = 0;
            end;
            else ascii_hex_flag = 0;
        unsave regs;
    end getbyte;
```



```
;**************************************************************************
; This routine retrieves the next two ascii characters in multimeter_dat
; converts them into a single hexadecimal value and return it via the
; accumulator.  Should either of the characters be something other than
; 0-9, a-f or A-F; the ascii_hex_flag is cleared.  In this way it is alm
; identical to the previous routine, except for the location that the
; data is retrieved from.  This routine is used by multimeter_func.
;**************************************************************************

Pseudocode:

get_data:
        save regs;
        accum = multimeter_data;
        call a_to_h(accum);
        if (accum <> $FF)
            then do;
                temp = accum << 4;
                accum = multimeter_data + 1;
                call a_to_h(accum);
                if (accum <> $FF)
                    then do;
                        accum = accum + temp;
                        ascii_hex_flag = 1;
                    end;
                    else ascii_hex_flag = 0;
            end;
            else ascii_hex_flag = 0;
        unsave regs;
    end get_data;

;**************************************************************************
; This routine takes the hexadecimal value in the accumulator, converts
; it into two ascii characters and places them into the output buffer.
; This routine assumes that the system interrupts are to remain disabled,
; so that it uses putchar_wo instead of putchar.
;**************************************************************************
```

```
Pseudocode:

putbyte_wo:
        call h_to_a(accum);
        call putchar_wo(ascii_chars+1);
        call putchar_wo(ascii_chars);
    end putbyte_wo;

;*****************************************************************************
; This routine takes the hexadecimal value in the accumulator, converts
; it into two ascii characters and places them into the output buffer.
; This routine wants to turn off the system interrupts so that the serial po
; handler cannot interfere so it uses putchar instead of putchar_wo.
;*****************************************************************************

Pseudocode:

putbyte:
        call h_to_a(accum);
        call putchar(ascii_chars+1);
        call putchar(ascii_chars);
    end putbyte;

;*****************************************************************************
; This routine converts the ascii character contained in the accumulator to
; the corresponding hex value which is returned in the accumulator.  If the
; character is something other than 0-9, a-f or A-F; $FF is returned instead
;*****************************************************************************

Pseudocode:

a_to_h:
        save regs;
        if (accum >= $30)
            then do;
                if (accum <= $39)
                    then accum = accum - $30;
                    else do;
                        accum = accum AND $DF;  /* force to upper case */
                        if ((accum >= $41) AND (accum <= $46))
                            then accum = accum - $37;
                            else accum = $FF;
                    end;
            end;
            else accum = $FF;
        unsave regs;
    end a_to_h;

;*****************************************************************************
; This routine converts the hexadecimal value contained in the accumulator
; into two ascii characters for transmission.  The two characters are stored
; in the word value, ascii_chars
;*****************************************************************************

Pseudocode:
```

```
h_to_a:
        save regs;
        temp = accum;
        call _h_nyb_to_a(accum);
        lsbyte of ascii_chars = accum;
        call _h_nyb_to_a(temp >> 4);
        msbyte of ascii_chars = accum;
        unsave reg;
    end h_to_a;


;**********************************************************************
; This captive subroutine converts the lower nybble of the accumulator
; into the corresponding ascii character and returns it in the accumulat
;**********************************************************************

Pseudocode:

    _h_nyb_to_a;
        accum = accum AND $0F;
        if (accum < $A)
            then accum = accum + $30;
            else accum = accum + $37;
    end _h_nyb_to_a;

;**********************************************************************
; This subroutine shifts the contents of dec_value into the accumulator
; 4 bits at a time, and converts the nybble into ascii by clearing the up
; nybble and adding $30.  It is assumed that only decimal digits will be
; in dec_value.
;**********************************************************************

Pseudocode:

shift_L4:
        shift dec_value << 4;
        place 4 msb's into accum;
        accum = accum AND $0F;
        accum = accum OR $30;
    end shift_L4;

;**********************************************************************
; This routine converts the 16 bit hexadecimal word contained in the cnvt
; into a packed decimal word that is returned in dec_value.
;**********************************************************************

Pseudocode:

h_to_d_word:
        save regs;
        dec_value = 0;
        mem_byte = (lsbyte of cnvt_data) AND $0F
        if (mem_byte <> 0)
            then dec_value = table_1[mem_byte];
        mem_byte = ((lsbyte of cnvt_data) AND $F0) >> 4;
```

```
        if (mem_byte <> 0)
            then dec_value = dec_value + table_16[mem_byte];
        mem_byte = (msbyte of cnvt_data) AND $0F
        if (mem_byte <>0)
            then dec_value = dec_value + table_256[mem_byte];
        mem_byte = ((msbyte of cnvt_data) AND $F0) >> 4;
        if (mem_byte <>0)
            then dec_value = dec_value + table_4096[mem_byte];
        unsave regs;
    end h_to_d_word;

;===========================================================================
;               HEXADECIMAL TO DECIMAL CONVERSION TABLES
;===========================================================================
;
table_1:
                .byte   $00, $01, $02, $03, $04, $05, $06, $07, $08
                .byte   $09, $10, $11, $12, $13, $14, $15
table_16:
                .word   $0000, $0016, $0032, $0048, $0064
                .word   $0080, $0096, $0112, $0128, $0144
                .word   $0160, $0176, $0192, $0208, $0224
                .word   $0240
table_256:
                .word   $0000, $0256, $0512, $0768, $1024
                .word   $1280, $1536, $1792, $2048, $2304
                .word   $2560, $2816, $3072, $3328, $3584
                .word   $3840
table_4096:
                .byte   $00,$00,$00, $96,$40,$00, $92,$81,$00
                .byte   $88,$22,$01, $84,$63,$01, $80,$04,$02
                .byte   $76,$45,$02, $72,$86,$02, $68,$27,$03
                .byte   $64,$68,$03, $60,$09,$04

;***************************************************************************
; This routine is used to convert the packed decimal byte in the accumulator
; into its hex equivalent.  If either nybble is > $9, the over_10_flag is se
; The result is returned via the accumulator.  To accomplish this, the most
; significant nybble is multiplied by 10 and added to the least significant
; nybble.
;
;       Since 10X = (2 + 8)X = 2X + 8X, it follows that:
;             10X = X(shifted left by 1) + X(shifted left by 3)
;                trick: 10 * X = (8 * X) + (2 * X)
;***************************************************************************

Pseudocode:

d_to_h:
        save regs;
        temp1 = accum;
        over_10_flag = 0;
        accum = accum AND $F0;
        if (accum < $A0)
            then do;
                temp2 = (accum >> 1) + (accum >> 3);
```

```
display_digits:
        save regs;
        dsply_port = #dsply_clk_off OR dsply_sel_dsb;
        dsply_port = dsply_port AND #dsply_sel_enb;
        do i = 0 to 4
            temp = digits[i];
            do j = 0 to 7
                if (digits[i] AND #$01)
                    then dsply_port = dsply_port OR #send_1_bit;
                    else dsply_port = dsply_port AND #send_0_bit;
                digits[i] = digits[i] >> 1;
                dsply_port = dsply_port OR #dsply_clk_on;
                dsply_port = dsply_port AND #dsply_clk_off;
            end;
            digits[i] = temp;
        end;
        dsply_port =dsply_port OR #dsply_sel_dsb;
        i_o_flags = i_o_flags AND #lcd_dsply_dun;
        unsave regs;
    end display_digits;


;**************************************************************************
; This routine will set up the array display_word with "blanking" informa
; in all digits, and then call display_hndlr() in order to clear the LCD
; display.
;**************************************************************************

Pseudocode:

display_clr:
        save regs;
        display_word[4] = 0;
        display_word[3] = #$80;
        display_word[2] = #$80;
        display_word[1] = #$80;
        display_word[0] = #$80;
        call display_hndlr;
        unsave regs;
    end display_clr;

;**************************************************************************
; This routine, called from the terminal, allows the user to set one digit
; of the LCD display to a specified character, while blanking the rest of
; the display.  The accessible digits are the rightmost 4, and they are
; number 1 - 4 starting from the right.
;**************************************************************************

Pseudocode:

set_digit:
        save regs;
        if ((command_buffer[1] < $31) OR (command_buffer[1] > $34))
            then call msg_hndlr('INVALID LCD DIGIT NUMBER');
            else do;
```

```
                i = command_buffer[1] - #$31;
                if ((command_buffer[2] < $30) OR ((command_buffer[2] > $39)
                        AND (command_buffer[2] < $41)) OR
                               (command_buffer[2] > $46))
                    then call msg_hndlr('DIGIT OUT OF RANGE');
                    else do;
                        j = command_buffer[2] - #$30;
                        if(j > $09)
                            then j = j - #$07;
                        multimeter_flag = 0;
                        do k = 3 to 0 step -1;
                            display_word[k] = #$80;
                            display_word[4] = 0;
                            display_word[i] = j;
                            call display_hndlr;
                        end;
                    end;
            end;
        unsave reg;
    end set_digit;
```

```
*****************************************************************************
 This routine is an unsigned multiply that takes the two 16 bit words in
 mult1 and mult2 and produces a 32 bit product which is returned,
 surprisingly enough, in "product".  The technique used is the commonly
 used shift and add algorithm.
*****************************************************************************
```

seudocode:

```
p_mult:
       save regs;
       product = 0;              { Zero out product memory location }
       scratch = mult2;          { Load multiplicand into lower 2 bytes }
                                 { of scratch area in memory }
                                 { Zero out upper two bytes of scratch area }
       do y_index = 0 to 1;      { For both lower & upper byte of multiplier }
           accumulator = mult1 + y_index;   { get the multiplier byte }
           do  x_index = 8 to 0 step -1     { For each bit in the byte  }
               accum = accum >> 1;          { shift it right to see if bit }
                                            { is set.... }
               if (carry_bit = 1)           { If so, then ...}
                   then do;
                       product = product + scratch;
                                            { add (shifted) multiplicand }
                                            { to product }
                       scratch << 1;        { In any case, shift the }
                   end;                     { go to next bit of multiplier }
           end;                             { get next byte of multiplier }
       end;
       unsave regs;
    end mp_mult;
```

```
*****************************************************************************
 This routine is au unsigned multiprecision division where the 32 bit
 dividend is stored in, get this, dividend; and the 16 bit divisor is
```

```
; stored in divisor.  The algorithm used was a "brute force" shift and
; subtract.  The quotient is returned...yep, you guess it...in quotient.
; As for the remainder, well you get the picture.  To save space in zerc
; page, quotient and remainder overlay dividend.
; In order to test to see if there will be an overflow, i.e. if the quot
; will be bigger than 16 bits we compare the divisor with the upper two
; of the dividend.  If the dividend is less, then we're OK, since the up
; two bytes can be no more than 32768 * the divisor and the result of th
; divsion must then be less than 32768 (16-bits, max.).
;*****************************************************************************

Pseudocode:

mp_div:
        save regs;
        if divisor = 0
            then set_div_by_flag;
            else do;
                if (dividend > $FFFF * divisor)
                    then set_div_overflow_flag;
                    else do;
                        tempquot = 0;   { zero out temporary quotient ar
                        tempdivr = divisor * $10000;
                                 { upper two bytes of tempdivr get divis
                                 { and lower two bytes get 0 }
                        for x_index = 2 to 1 step -1  { Do two bytes wor
                            for bit_cntr = 8 to 1 step -1
                                                 { Do a shift for each bi
                                tempqout << 1;
                                tempdivr >> 1;
                                tempdiff = dividend - tempdivr;
                                                 { Long word operation }
                                if (tempdiff >= 0)
                                    then do;
                                        tempquot = tempquot + 1;
                                        dividend = tempdiff;
                                                 { Make subtraction real
                                    end;
                            end;
                        end;
                    quotient = tempquot;{ load up quotient in correct ar
                    end;
            end;
        unsave regs;
    end mp_div;

;*****************************************************************************
; This routine reads the contents of the Time of Day counters in the
; Am9513 timer and stores the hours and minutes information away in zero
; locations "minutes" and "hours" for use by out_time and put_time.
;*****************************************************************************

Pseudocode:

find_time:
        save regs;
```

```
        timer_csr = #$A3;
        timer_csr = #$19;
        i = timer_data;
        i = i OR timer_data;
        if (i = 0)
            then do;
                timer_csr = #$A2;
                timer_csr = #$1A;
            end;
        minutes = timer_data;
        hours = timer_data;
        unsave regs;
    end find_time;
```

```
****************************************************************************
 This routine reads the contents of the zero-page locations "minutes" and
 "hours" and formats the information for display on the LCD display.
****************************************************************************
```

```
seudocode:

ut_time:
        save regs;
        display_word = minutes AND #$0F;
        display_word+1 = ((minutes AND #$F0) >> 4);
        display_word+2 = hours AND #$0F;
        display_word+3 = ((hours AND #$F0) >> 4);
        display_word+4 = #$02;
        call display_hndlr;
        unsave regs;
    end out_time;
```

```
****************************************************************************
 This routine locates the "raw A/D count" for a specified channel and
 converts it into decimal volts and formats the value for the front panel
 display.
****************************************************************************
```

```
suedocode:

ut_volts:
        save regs;
        mem_byte = get_data();
        if (ascii_hex_flag = 1)
            then do;
                call find_volts;
                if (error_flag = 0)
                    then do;
                        dec_value = h_to_d_word(cnvt_data);
                        if (dec_value >= 10000)
                            then do;
                                accum = $03;
                                call display_error;
                            end;
                            else do;
```

```
                                do index = 3 to 0 step -1;
                                    call shift_L4;
                                    display_word[index] = accum;
                                end;
                            end;
                        display_word[4] = dot_position; /* this includes
                                                           sign flag dat
                        call display_hndlr;
                    end;
            end;
            else call display_huh;
        unsave reg;
    end out_volts;


;*************************************************************************
; This is the subroutine which actually does the location of the "raw A/I
; count" for a specified channel from the dump_array, put_volts or out_v(
;*************************************************************************

Psuedocode:

find_volts:
        save regs;
        if (mem_byte <= $39)
            then do;
                if ((mem_byte = $38) OR (mem_byte = $39))
                    then dot_position = 3;
                    else dot_position = 1;
                mem_byte = voltage_channels[mem_byte];
                if (mem_byte <> $FF)
                    then do;
                        error_flag = 0;
                        cnvt_data = dump_array[mem_byte*2];
                        if (cnvt_data < 0)
                            then do;
                                cnvt_data = 2's comp (cnvt_data);
                                sign_flag = 1;
                            end;
                            else sign_flag = 0; /* sign flag is in msb of
                                                   dot_position */
                        cnvt_data = cnvt_data << 2;
                    end;
                    else error_flag = 1;
            end;
            else error_flag = 1;
        unsave regs;
    end find_volts;

;=========================================================================
;                     USER VOLTAGE CHANNNEL LOOKUP TABLE
;=========================================================================

voltage_channels:
                .byte   $00, $0C, $0D, $0E, $0F, $10, $11, $12
                .byte   $13, $14, $FF, $FF, $FF, $FF, $FF, $FF
```

```
                .byte   $15, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $16, $17, $18, $19, $1A, $FF, $FF
                .byte   $22, $23

;***************************************************************************
; This routine locates the "raw A/D count" for a specified channel and
; converts it into decimal ampss and formats the value for the front panel
; display.
;***************************************************************************

Pseudocode:

out_amps:
        save regs;
        mem_byte = get_data();
        if (ascii_hex_flag = 1)
            then do;
                if (mem_byte = $40)
                    then do;
                        cnvt_data = abs_total_chgr_I;
                        dot_position = 1;
                        if (total_chgr_I < 0)
                            then sign_flag = 1;
                        dec_value = bf_div10(cnvt_data);
                    end;
                    else do;
                        call find_amps;
                        if (error_flag = 0)
                            then dec_value = h_to_d_word(cnvt_data);
                    end;
                if (dec_value >= 10000)
                    then do;
                        error_num = $03;
                        call display_error;
                    end;
                    else do;
                        do index = 3 to 0 step -1;
                            call shift_L4;
                            display_word[index] = accum;
                        end;
                        display_word[4] = dot_position;
                        call display_hndlr;
                    end;
            end;
            else call display_huh;
        unsave reg;
    end out_amps;

 ***************************************************************************
  This is the subroutine which actually does the location of the "raw A/D
  count" for a specified channel from the dump_array, put_amps or out_amps.
 ***************************************************************************
```

```
Pseudocode:

find_amps:
        save regs;
        if (mem_byte <= $35)
            then do;
                mem_byte = current_channels[mem_byte];
                if (mem_byte <> $FF)
                    then do;
                        if ((mem_byte <= $10) AND (mem_byte <> 0)
                            then dot_position = 2;
                            else dot_position = 1;
                        error_flag = 0;
                        cnvt_data = dump_array[mem_byte*2];
                        if (cnvt_data < 0)
                            then do;
                                cnvt_data = 2's comp(cnvt_data);
                                sign_flag = 1;  /* sign bit is ms bit o
                                                   dot_position */
                            end;
                            else sign_flag = 0;
                    end;
                    else error_flag = 1;
            end;
            else error_flag = 1;
        unsave regs;
    end find_amps;

;=============================================================================
;                        USER CURRENT CHANNNEL LOOKUP TABLE
;=============================================================================

current_channels:
                .byte   $01, $02, $03, $04, $05, $06, $07, $08
                .byte   $09, $0A, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $0B, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF
                .byte   $FF, $1B, $1C, $1D, $1E, $1F

;*****************************************************************************
; This routine displays or arranges for the display of 6 data channels t
; the front panel LCD display.  The data channels are:
;       channel 00 - state of charge
;       channel 36 - battery temp
;       channel 37 - freezer temp
;       channel 40 - corrected state of charge
;       channel 41 - equalization count
;       channel 42 - pwm_value
;*****************************************************************************

Pseudocode:

out_data_channel:
        save regs;
```

```
      mem_byte = get_data();
      if (ascii_hex_flag = 1)
        then do;
          if (mem_byte = 0) then call out_soc;
            else if (mem_byte = $40) then call out_csoc;
               else if (mem_byte = $41) then call out_equal;
                  else if (mem_byte = $42) then call out_pwm;
                     else do;
                       call find_temp;
                       if (error_flag = 0)
                         then do;
                           if ((div_by_zero = 0) AND (div_overflow) = 0)
                             then do;
                               dec_value = h_to_d_word(cnvt_data);
                               if (dec_value >= 10000)
                                 then do;
                                   accum = $06;
                                   call display_error;
                                 end;
                                 else do;
                                   do index = 3 to 0 step -1;
                                     call shift_L4;
                                     display_word[index] = accum;
                                   end;
                                   display_word[4] = dot_position;
                                   call display_hndlr;
                                 end;
                             end;
                             else do;
                               error_num = $06;
                               call display_error;
                             end;
                     end;
                     else call display_huh;
        end;
        else call display_huh;
      unsave regs;
  end out_data_channel;

;*****************************************************************************
 This routine locates the msbyte of state_of_chg, cstate_of_chg, pwm_value,
 or equalization count (depending on the entry point), converts it into a
 percentage and returns the value in dec_value ready for shifting and output

      State_of_chg is stored:  X.XXX XXXX XXXX XXXX
                                ^
                      implied binary point

      Where this value represents (decimal percent)/100) * 32768

 To convert back to decimal for display, we first multiply by 200 (base 10)
 resulting in:

 (decimal percentage/100) * 32768 * 2 * 100 =  (decimal percentage) * 65566

 So, by taking the msbyte of the result, rounding it up or down by looking
```

```
; at the msb of the previous byte, and converting to decimal we have a v
; for output.  Although the pseudocode shows this to be 5 separate routi
; in the assembly code they are condensed to just one, with 4 entry poin
;************************************************************************

Pseudocode:

out_csoc:
        save regs;
        msbyte of mult1 = msbyte of cstate_of_chg;
        lsbyte of mult1 = $80;
        call _display_soc();
        unsave regs;
   end out_csoc;

 out_soc:
        save regs;
        msbyte of mult1 = msbyte of state_of_chg;
        lsbyte of mult1 = 2nd msbyte of state_of_chg;
        call _display_soc();
        unsave regs;
   end out_soc;

 out_equal:
        save regs;
        msbyte of mult1 = msbyte of equal_count;
        lsbyte of mult1 = 2nd msbyte of equal_count;
        call _display_soc();
        unsave regs;
   end out_equal;

 out_pwm:
        save regs;
        msbyte of cnvt_data = 0;
        lsbyte of cnvt_data = pwm_value >> 2;
        call _display_pwm:
        unsave regs;
   end out_pwm;

_display_soc: procedure;
        mult2 = 200;
        call mp_mult;
        cnvt_data = product[2];
        if (product[1] AND bit 7 <> 0)
            then cnvt_data = cnvt_data + 1;  /* round up if necessary */
_display_pwm:
        dec_value = h_to_d_word(cnvt_data);
        display_word[4] = 0;
        display_word[3] = $80;
        call shift_L4;
        call shift_L4;
        if (accum = $30)
            then display_word[2] = $80;
            else display_word[2] = accum;
        call shift_L4;
        display_word[1] = accum;
```

```
       call shift_L4;
       display_word[0] = accum;
       call display_hndlr;
    end _display_soc;

***************************************************************************
 This routine calls the proper routines to retrieve the channel current
 and voltage, multiply them and places the result to the serial output
 buffer.
***************************************************************************

seudocode:

ut_watts:
       save regs;
       call do_p_watts;
       if (error_flag = 0)
           then do;
               if (dot_position AND $80 <> 0)
                   then call putchar('-');
               call putchar(msbyte of dec_value + $30);
               call shift_L4;
               call putchar(accum);
               call shift_L4;
               call putchar(accum);
               call shift_L4;
               call putchar(accum);
               call shift_L4;
               call putchar(accum);
               call msg_hndlr ('WATTS');
           end;
           else call msg_hndlr('INVALID CHANNEL OR DATA');
       unsave regs;
    end put_watts;

***************************************************************************
 This routine calls the proper routines to retrieve the channel current
 and voltage, multiply them and sets up the result for the front panel
 LCD display.  Notice that the maximum power that can be displayed is
 19999 watts.
***************************************************************************

seudocode:

ut_watts:
       save regs;
       call do_o_watts;
       if (error_flag = 0)
           then do;
               if (dec_value >= 20000)
                   then do;
                       error_num = $03;
                       call display_error;
                   end;
                   else do;
                       do index = 3 to 0 step -1;
```

```
                              call shift_L4;
                              display_word[index] = accum;
                          end;
                          if (msbyte of dec_value = 1)
                              then display_word[4] = dot_position OR $40;
                              else display_word[4] = dot_position;
                          call display_hndlr();
                      end;
              end;
              else call display_huh;
          unsave regs;
      end out_watts;

;*************************************************************************
; This routine does the actual retrieving, multiplying and conversion tc
; "displayable watts" for out_watts and put_watts.
;*************************************************************************

Pseudocode:

find_watts:
          save regs;
          sign_flag = 0;
          if (mem_byte <= 10)
             then do;
                  save_watt_ch_num = mem_byte;
                  mem_byte = 0;       /* this forces battery_V to be used f
                                         the calculation */
                  call find_volts;
                  mem_byte = save_watt_ch_num;
              end;
              else call find_volts;
          if (error_flag = 0)
              then do;
                  volt_sign = dot_position;  /* get volt sign */
                  mult1 = cnvt_data;
                  call find_amps;
                  if (error_flag = 0)
                      then do;
                          dot_position = (dot_position XOR volt_sign) AND $
                                                      /* include amp sign *,
                          mult2 = cnvt_data;
                          call mp_mult;
                          dividend = product;
                          if ((mem_byte >= 31) OR (mem_byte = 0))
                              then divisor = 100;
                              else divisor = 1000;
                          call mp_div;
                          cnvt_data = quotient;
                          dec_value = h_to_d_word(cnvt_data)
                      end;
              end;
          unsave regs;
      end find_watts;

;*************************************************************************
```

```
; This routine retrieves the channel number, tests to see that it is
; acceptable and calls find_watts if it is.  Otherwise it sets the error_fla
; Both find_amps and find_volts test for channels, but there are couple othe
; channels, such as channel 38 that are not meaningful in this context.  Hen
; the double-check.  Notice, that although the pseudocode implies that there
; 3 routines, they are actually written as one routine with 2 entry points.
;***************************************************************************

Pseudocode:

do_p_watts:
        save regs;
        cmd_out_ptr = 1;
        mem_byte = getbyte();
        call _do_watts;
        unsave regs;
    end do_p_watts;

 do_o_watts:
        save regs;
        mem_byte = get_data();
        call _do_watts;
        unsave regs;
    end do_o_watts;

    _do_watts: procedure;
        error_flag = 0;
        if (ascii_hex_flag = 1)
            then do;
                if (mem_byte < 36)
                    then call find_watts;
                    else error_flag = 1;
            end;
            else error_flag = 1;
        unsave regs;
    end do_watts;

 **************************************************************************
  This routine routes or arranges for the routing of 6 data channels to
  the serial port.  The data channels are:
        channel 00 - state of charge
        channel 36 - battery temp
        channel 37 - freezer temp
        channel 40 - corrected state of charge
        channel 41 - equalization count
        channel 42 - pwm_value
 **************************************************************************

 seudocode:

 ut_data_channel:
        save regs;
        cmd_out_ptr = 1;
        mem_byte = getbyte();
        if (ascii_hex_flag = 1)
          then do;
```



```
            if (mem_byte = 0) then call put_soc;
              else if (mem_byte = $40) then call put_csoc;
                 else if (mem_byte = $41) then call put_equal;
                    else if (mem_byte = $42) then call out_pwm;
                else call dump_temp;
          end;
          else call put_huh;
        unsave regs;
    end put_data_channel;

;***************************************************************************
; This routine locates the msbyte of state_of_chg, cstate_of_chg, pwm_va
; or equalization count (depending on the entry point), converts it intc
; percentage and returns the value in dec_value ready for shifting and c
;
;       State_of_chg is stored:  X.XXX XXXX XXXX XXXX
;                                 ^
;                              implied binary point
;
;       Where this value represents (decimal percent)/100) * 32768
;
; To convert back to decimal for display, we first multiply by 200 (base
; resulting in:
;
; (decimal percentage/100) * 32768 * 2 * 100 =  (decimal percentage) * 6
;
; So, by taking the msbyte of the result, rounding it up or down by look
; at the msb of the previous byte, and converting to decimal we have a v
; for output.  Although the pseudocode shows this to be 5 separate routi
; in the assembly code they are condensed to just one, with 4 entry poin
;***************************************************************************

Pseudocode:

put_csoc:
        save regs;
        msbyte of mult1 = msbyte of cstate_of_chg;
        lsbyte of mult1 = $80;
        call put_soc();
        unsave regs;
   end_csoc;

put_soc:
        save regs;
        msbyte of mult1 = msbyte of state_of_chg;
        lsbyte of mult1 = 2nd msbyte of state_of_chg;
        call put_soc();
        unsave regs;
   end_soc;

put_equal:
        save regs;
        msbyte of mult1 = msbyte of equal_count;
        lsbyte of mult1 = 2nd msbyte of equal_count;
        call _display_soc();
        unsave regs;
```

```
  end put_equal;

ut_pwm:
        save regs;
        msbyte of cnvt_data = 0;
        lsbyte of cnvt_data = pwm_value >> 2;
        call _put_pwm:
        unsave regs;
  end put_pwm;

out_soc: procedure;
        mult2 = 200;
        call mp_mult;
        cnvt_data = product[2];
        if (product[1] AND bit 7 <> 0)
            then cnvt_data = cnvt_data + 1;  /* round up if necessary */
out_pwm:
        dec_value = h_to_d_word(cnvt_data);
        call shift_L4;
        call shift_L4;
        if (accum = $30) then accum = $20;
        call putchar(accum);
        call shift_L4;
        call putchar(accum);
        call shift_L4;
        call putchar(accum);
        call putchar('%');
   end _put_soc;

************************************************************************
 This routine locates the "raw A/D count" for a specified channel and
 converts it into decimal volts and formats the data for the serial
 port.
************************************************************************

seudocode:

ut_volts:
        save regs;
        cmd_out_ptr = 1;
        mem_byte = getbyte();
        if (ascii_hex_flag = 1)
            then do;
mp_volts:
                call find_volts;
                if (error_flag = 0)
                    then do;
                        dec_value = h_to_d_word(cnvt_data);
                        if (dec_value >= 10000)
                            then call msg_hndlr('READING OUT OF RANGE');
                            else do;
                                if (sign_flag = 1)
                                    then do;
                                        call putchar('-');
                                        sign_flag = 0;
                                    end;
```

```
                                          else call putchar(' ');
                                      call shift_L4;
                                      call putchar(accum);
                                      if (dot_position = 3)
                                          then call putchar('.');
                                      call shift_L4;
                                      call putchar(accum);
                                      call shift_L4;
                                      call putchar(accum);
                                      if (dot_position = 1)
                                          then call putchar('.');
                                      call shift_L4;
                                      call putchar(accum);
                                      call msg_hndlr('volts');
                                  end;
                          end;
                          else call put_huh;
                  end;
                  else call put_huh;
              unsave reg;
          end put_volts;

;************************************************************************
; This routine locates the "raw A/D count" for a specified channel and
; converts it into decimal amps and formats the data for the serial
; port.
;************************************************************************

Pseudocode:

put_amps:
        save regs;
        cmd_out_ptr = 1;
        mem_byte = getbyte();
        if (ascii_hex_flag = 1)
            then do;
                if (mem_byte = $40)
                    then call put_total_I;
                    else do;
dump_amps:
                        call find_amps;
                        if (error_flag = 0)
                            then do;
                                dec_value = h_to_d_word(cnvt_data);
                                if (dec_value >= 10000)
                                    then call msg_hndlr('READING OUT OF R
                                    else do;
                                        if (sign = 1) then call putchar (
                                        call shift_L4;
                                        call putchar(accum);
                                        call shift_L4;
                                        call putchar(accum);
                                        if (dot_position = 2)
                                            then call putchar('.');
                                        call shift_L4;
                                        call putchar(accum);
```

```
                                        if (dot_position = 1)
                                            then call putchar('.');
                                        call shift_L4;
                                        call putchar(accum);
                                        call msg_hndlr('amps');
                                    end;
                            end;
                            else call put_huh;
                    end;
            else call put_huh;
        unsave reg;
    end put_amps;
```

```
****************************************************************************
 This routine takes the raw channel number from the command_buffer and
 uses it as an offset into the dump_array and outputs the contents in
 ascii format to the serial port.
****************************************************************************
```

seudocode:

```
ut_raw_channel:
        save regs;
        cmd_out_ptr = 1;
        offset = getbyte();
        if (ascii_hex_flag = 0) then call put_huh;
            else do;
                if (offset > $37) then call put_huh;
                    else do;
                        if (offset = $37) then offset = $36;
                        offset = d_to_h(offset);
                        offset = (offset*2) + 1;
                        mem_byte = dump_array[offset];
                        call putbyte(mem_byte);
                        offset = offset - 1;
                        mem_byte = dump_array[offset];
                        call putbyte(mem_byte);
                    end;
            end;
        cmd_out_ptr = 0;
        unsave regs;
    end put_raw_channel;
```

```
****************************************************************************
 This routine outputs the current time to the serial output port.  It is
 called by itself and also from dump_state.
****************************************************************************
```

seudocode:

```
ut_time:
        save regs;
        call msg_hndlr('TIME:');
        call putbyte(hours);
        call putchar(':');
        call putbyte(minutes);
```

```
        call putchar('CR');
        call putchar('LF');
    end put_time;


;**********************************************************************
; This routine collects the "raw A/D count" for the specified channel,
; converts it into a decimal temperature, format it and outputs it to th
; serial port.
;**********************************************************************

Pseudocode:

put_temp:
        save regs;
        cmd_out_ptr = 1;
        mem_byte = getbyte();
        if (ascii_hex_flag = 1)
            then do;
dump_temp:
                call find_temp;
                if (error_flag = 0)
                    then do;
                        if (div_by_zero = 1)
                            then call msg_hndlr('DIVIDE BY 0');
                        if (div_overflow = 1)
                            then call msg_hndlr('DIVIDE OVERFLOW');
                        dec_value = h_to_d_word(cnvt_data);
                        if (dec_value >= 10000)
                            then call msg_hndlr('READING OUT OF RANGE');
                            else do;
                                if (sign_flag = 1)
                                    then call putchar('-');
                                call shift_L4;
                                call putchar(accum);
                                call shift_L4;
                                call putchar(accum);
                                call shift_L4;
                                call putchar(accum);
                                call putchar('.');
                                call shift_L4;
                                call putchar(accum);
                                call msg_hndlr('DEG C');
                            end;
                    end;
                    else call put_huh;
            end;
            else call put_huh;
        unsave reg;
    end put_temp;

;**********************************************************************
; This routine is where the actual look up and conversion to degrees of t
; "raw A/D count" is accomplished.  The final value is actually 10*Ts to
; permit display of temperature to 1 decimal place. Only channels 36 and :
; are considered valid.
```

```
;****************************************************************************

Pseudocode:

find_temp:
        save regs;
        error_flag = 0;
        sign_flag = 0;
        dot_position = 1;
        if ((mem_byte = $36) OR (mem_byte = $37))
            then do;
                mem_byte = mem_byte - $16;
                cnvt_data = dump_array[mem_byte];
                cnvt_data = cnvt_data << 2;
                cnvt_data = cnvt_data - 2560;
                if (cnvt_data < 0)
                    then do;
                        cnvt_data = 2's comp (cnvt_data);
                        sign_flag = 1;
                    end;
                strobe watchdog timer;
                cnvt_data = cnvt_data * 2092;
                cnvt_data = cnvt_data / 10000;
                if (sign_flag = 1)
                    then cnvt_data = 250 + cnvt_data;
                    else do;
                        cnvt_data = 250 - cnvt_data;
                        if (cnvt_data < 0)
                            then do;
                                cnvt_data = 2's comp (cnvt_data);
                                sign_flag = 1;
                            end;
                            else sign_flag = 0;
                    end;
            end;
            else error_flag = 1;
    end find_temp;

****************************************************************************
 This routine sets the Time of Day counters to a new value (i.e. sets the
 clock).  User input is checked for two types of errors: 1) if an entry
 is not between 0 and 9; and 2) if the hours value > 23 or the minutes
 value > 59.  If error type 1 is encountered, an Er04 is signalled to
 the LCD or the message "Bad digit value for set_tim\n" is sent to the
 terminal.  If error type 2 is encountered, an Er05 is signalled to the
 LCD display or the message "Invalid hours/minutes value" is sent to the
 terminal.  This routine has two entry points.  Which one is used depends
 upon which input device generated the command, terminal keyboard, or
 front panel keypad.  Although the pseudocode indicates 3 routines, the
 assembly code is actually 1 routine with 2 entry points.
 ***************************************************************************

 eudocode:

 t_tim_term:
        cflag = 0;
```

```
        _set_tim1();
    end set_tim_term;

set_tim_kybd:
        cflag = 1;
        _set_tim1();
    end set_tim_kybd;

_set_tim1:
        save regs:;
        i = 1 + cflag;
        minutes = 0;
        hours = 0;
        k = (hours << 8) OR minutes;
        do j = 3 to 0 step -1;
            if((command_buffer[i] < $30) OR (command_buffer[i] > $39))
                then do;
                    if (cflag = 1)
                        then call display_error(#$04);
                        else call msg_hndlr('DIGIT OUT OF RANGE');
                    goto set_time_done;
                end;
                else do;
                    k = k << 4;
                    k = k OR (command_buffer[i] - #$30);
                    i = i + 1;
                end;
        end;
        minutes = k AND #$FF;
        hours = (k >> 8) AND #$FF;
        if ((hours > 23) OR (minutes > 59))
            then do;
                if (cflag == 1)
                    then call display_error(#$05);
                    else call msg_hndlr('INVALID HOURS/MINUTES VALUE');
            end;
            else do;
                timer_csr = #$C3;
                timer_csr = #$09;
                timer_data = #$00;
                timer_data = #$00;
                timer_csr = #$0A;
                timer_data = #$00;
                timer_data = #$00;
                timer_csr = #$43;
                timer_csr = #$0A;
                timer_data = minutes;
                timer_data = hours;
                timer_csr = #$0F;
                if ((minutes >= #$00) AND (minutes < #$30))
                    then do;
                        timer_data = #$30;
                        timer_data = hours;
                    end;
                    else do;
                        timer_data = #$00;
```

```
                        timer_data = hours + 1;
                    end;
                timer_csr = #$43;
                timer_csr = #$0A;
                timer_data = #$00;
                timer_data = #$00;
                timer_csr = #$23;
            end;
    set_time_done:
        unsave regs;
    end _set_tim1;
```

****
This routine which can be called from both the front panel keypad and the serial port, sets the load shed threshold of the indicated load to the level specified.
****

seudocode:

```
et_load_shed:
       save regs;
       mem_byte = command_buffer[1];
       load_num = a_to_h(mem_byte);
       if ((load_num <> $FF) AND (load_num <= 5) AND (load_num <> 0))
           then do;
               cmd_out_ptr = 2;
               prcnt = find_percent();
               if (accum <> $FF)
                   then shed_thresh[load_num-1] = msbyte of prcnt;
                   else call put_huh;
           end;
           else if (kybd_cmd_flag = 1)
               then call display_huh;
               else call put_huh;
       unsave regs;
   end set_load_shed;
```

****
This routine which can be called from both the front panel keypad and the serial port, sets the load restoration threshold of the indicated load to the level specified.
****

seudocode:

```
et_load_restor:
       save regs;
       mem_byte = command_buffer[1];
       load_num = a_to_h(mem_byte);
       if ((load_num <> $FF) AND (load_num <= 5) AND (load_num <> 0))
           then do;
               cmd_out_ptr = 2;
               prcnt = find_percent();
               if (accum <> $FF)
                   then restor_thresh[load_num-1] = msbyte of prcnt;
```

```
                    else call put_huh;
            end;
            else if (kybd_cmd_flag = 1)
                then call display_huh;
                else call put_huh;
        unsave regs;
    end set_load_restor;


;*************************************************************************
; This routine which can be called from both the front panel keypad and th
; serial port, sets the initial battery state of charge to  the level spec
;*************************************************************************

Pseudocode:

set_init_soc:
        save regs;
        cmd_out_ptr = 1;
        prcnt = find_percent();
            if (accum <> $FF)
                then do;
                    2 msbyte's of state_of_chg = prcnt;
                    3 lsbyte's of state_of_chg = 0;
                end;
                else if (kybd_cmd_flag = 1)
                    then call display_huh;
                    else call put_huh;
        unsave regs;

;*************************************************************************
; This routine is used by the pervious 3 routines to convert the 3 charact
; in the command buffer starting with the position pointed to by cmd_in_pt
; into the fraction corresponding to the percentage of state of charge.
; This value is returned in percent.  For example,
;
;                 100 percent results in a value of %1000 0000 0000 0000
;                  75 percent results in a value of %0110 0000 0000 0000
;                  50 percent results in a value of %0100 0000 0000 0000
;                  25 percent results in a value of %0010 0000 0000 0000
;                   1 percent results in a value of %0000 0001 0100 0111
;
; If the 3 characters represent a value over 100% or contain digits over
; 9, $FF is returned in the accumulator.
;*************************************************************************

Pseudocode:

find_percent:
        save regs;
        temp = cmd_out_ptr;
        cmd_out_ptr = cmd_out_ptr + 1;
        mem_byte = getbyte();
        if (ascii_hex_flag = 1)
            then do;
                mem_byte = d_to_h(mem_byte);
```

```
              mem_byte = mem_byte * 2;
              if (over_10_flag = 0)
                  then do;
                      if ((mem_byte = 0) AND (command_buffer[temp] = $31))
                          then prcnt = percent_tbl[$C8];
                          else if ((mem_byte <> 0) AND
                                                   (command_buffer[temp] = #30))
                               then prcnt = percent_tbl[mem_byte];
                               else accumulator = $FF;
                  end;
                  else accumulator = $FF;
          end;
          else accumulator = $FF
      unsave regs;
  end find_percent;

;==========================================================================
;             PERCENT TABLES FOR FIND_PERCENT
;==========================================================================

percent_tbl:
      .word $0000, $0147, $028F, $03D7, $051E, $0666, $07AE, $08F5
      .word $0A3D, $0B85, $0CCC, $0E14, $0F5C, $10A3, $11EB, $1333
      .word $147A, $15C2, $170A, $1851, $1999, $1AE1, $1C28, $1D70
      .word $1EB8, $2000, $2147, $228F, $23D7, $251E, $2666, $27AE
      .word $28F5, $2A3D, $2B85, $2CCC, $2E14, $2F5C, $30A3, $31EB
      .word $3333, $347A, $35C2, $370A, $3851, $3999, $3AE1, $3C28
      .word $3D70, $3EB8, $4000, $4147, $428F, $43D7, $451E, $4666
      .word $47AE, $48F5, $4A3D, $4B85, $4CCC, $4E14, $4F5C, $50A3
      .word $51EB, $5333, $547A, $55C2, $570A, $5851, $5999, $5AE1
      .word $5C28, $5D70, $5EB8, $6000, $6147, $628F, $63D7, $651E
      .word $6666, $67AE, $68F5, $6A3D, $6B85, $6CCC, $6E14, $6F5C
      .word $70A3, $71EB, $7333, $747A, $75C2, $770A, $7851, $7999
      .word $7AE1, $7C28, $7D70, $7EB8, $8000

;**************************************************************************
; This routine, called every 100 msec. by the run_task_master, provides the
; absolute values of battery_V and total_chgr_I for use by the max power
; tracking routine as well as several others.
;**************************************************************************

Pseudocode:

abs_cnvt:
      save regs;
      if (battery_V < 0)
          then abs_battery_V = -battery_V;
          else abs_battery_V = battery_V;
      if (total_chgr_I < 0)
          then abs_total_chgr_I = -total_chgr_I;
          else abs_total_chgr_I = total_chgr_I;
      unsave regs;
   end abs_cnvt;

;**************************************************************************
; This routine is called to dump the current total charger current as part
```

```
; of the dump_state routine.  The put_total_I entry point is used by ou
; data_channel and put_data_channel.
;*********************************************************************

Pseudocode:

dump_total_I:
        save regs;
        call msg_hndlr('TOT CHGR I =');
put_total_I:
        cnvt_data = abs_total_chgr_I;
        dec_value = bf_div10(cnvt_data);
        if (total_chgr_I < 0)
            then call put_char('-');
        call shift_L4;
        call putchar(accum);
        call shift_L4;
        call putchar(accum);
        call shift_L4;
        call putchar(accum);
        call putchar('.')
        call shift_L4;
        call putchar(accum);
        call msg_hndlr('amps');
        unsave reg;
    end dump_total_I;

;*********************************************************************
; This routine does a BRUTE FORCE divide by 10 (base 10) of the number i
; cnvt_data and returns the quotient in dec_value because due to this cl
; algorithm, it is also converted into decimal at the same time(!).  All
; of this is necessary to convert the current in the form xx.xx to 0xx.x
;*********************************************************************

Pseudocode:

bf_div10:
        save regs;
        dec_value = 0;
        do while (cnvt_data > 1000 );               1000 (base10)
            cnvt_data = cnvt_data - $3E8;           1000 (base16)
            dec_value = dec_value + 100;
        end;
        do while (cnvt_data > 100 );                100 (base10)
            cnvt_data = cnvt_data - $64;            100 (base16)
            dec_value = dec_value + 10;
        end;
        do while (cnvt_data > 10 );                 10 (base10)
            cnvt_data = cnvt_data - $A;             10 (base16)
            dec_value = dec_value + 1;
        end;
        unsave regs;
    end bf_div_10;
```

```
XXXXXX          XXXXX   XXXXX    XXXX   X    X
X               X    X  X    X  X    X  XX  XX
XXXXX           X    X  X    X  X    X  X XX X
X               XXXXX   XXXXX   X    X  X    X
X               X       X   X   X    X  X    X
XXXXXX          X       X    X   XXXX   X    X
```

```
;***************************************************************************
; This routine, called at system reset, and from the serial port for te
; will turn on all segments of the LCD display, all warning LED indicat
; and the warning buzzer for a period of (about) 1 second.  After the t
; delay, the routine will return the LCD display, warning buzzer, and
; indicator LEDs to their original state before this routine was called
;***************************************************************************

Pseudocode:

lamp_test:
        save regs:;
        if ((compute_flags AND #run_flag) <> 0)
            then do;
                digits[4] = #dspy_sync_ctrl OR #$0F;
                digits[3] = #$FF;
                digits[2] = #$FF;
                digits[1] = #$FF;
                digits[0] = #$FF;
                call display_digits();
                led_out_latch = stow_leds OR #leds_on;
                disable interrupts;
                kybd_wr_port = ~(column_number AND #$0F) OR #bell_on;
                enable interrupts;
                do i = 2 to 0 step -1;          /* waste ~ 1 second */
                    do j = 255 to 0 step -1;
                        do j = 255 to 0 step -1;
                        end;
                    end;
                end;
                led_out_latch = stow_leds;
                disable interrupts;
                if(alarm_flags < 0)
                    then kybd_wr_port = ~(column_number AND #$0F) OR #b
                    else kybd_wr_port = ~(column_number AND #$0F);
                enable interrupts;
                call display_hndlr();
            end;
        unsave regs;
    end lamp_test;

;***************************************************************************
; This routine performs the "MULTIMETER FUNCTION".  Five of the front pa
; display functions operate continuously like a multimeter.  Each time t
; routine is called, which is once every 100 msec , it reactivates the c
; function whose starting address is stored in multimeter_addr. If
; multimeter_flag is zero, this routine is skirted.
;***************************************************************************

Pseudocode:
```

```
multimeter_func:
        save regs;
        if (multimeter_flag = 1)
            then do;
                call display_function @ multimeter_addr;
            end;
        unsave regs;
    end multimeter_func;

;**********************************************************************
; This routine "opens" memory at the indicated address and sends the
; contents to the serial port.  Successive CR's cause the next location
; be read, while "-" characters will back up to the previous location.
; Any location can be changed by entering the new data and hitting eithe
; a CR or a "-".  The CR will take you forward, while the "-" will take
; backwards.
;**********************************************************************

Pseudocode:

open_mem:
        save regs;
        cmd_out_ptr = 1;
        msbyte of mem_addr = getbyte();
        if (ascii_hex_flag = 0)
          then put_huh;
          else do;
             lsbyte of mem_addr = getbyte();
             if (ascii_hex_flag = 0)
                then put_huh;
                else do;
                   call _show_mem;
                   do while (cntrl_z_flag = 0);
                      do while (serial_cmd_flag = 0)
                         strobe watchdog timer;
                      end;
                      serial_cmd_flag = 0;
                      mem_byte = getbyte();
                      if ((ascii_hex_flag = 0) AND (command_buffer[0] <>
                         then do;
                            call put_huh;
                            cmd_out_ptr = 0;
                            command_buffer[0] = 0;
                         end;
                         else do;
                            if (command_buffer[0] <>0)
                                     then mem[mem_addr] = mem_byte;
                            if (look_bkward = 1)
                               then do;
                                  look_bkward = 0;
                                  decr mem_addr;
                                  call _show_mem;
                                  call msg_hndlr('CRLF');
                               end;
                               else do;
                                  incr mem_addr;
```

PRECEDING PAGE BLANK NOT FILMED

```
                                     call _show_mem;
                                  end;
                            end;
                      end;
                  end;
             end;
           cntrl_z_flag = 0;
           unsave regs;
       end open_mem;

 _show_mem:
           call putbyte(mem_addr+1);
           call putbyte(mem_addr);
           call putchar(' ');
           mem_byte = mem[mem_addr];
           call putbyte(mem_byte);
           cmd_out_ptr = 0;
       end _show_mem;

;****************************************************************************
; This routine is called from the 4 MSec interrupt handler to control keyboa
; scanning.  If no key is pressed, reset the debounce counter and skip to th
; next keyboard column. If a key is pressed, decrement the debounce counter,
; and if we've reached zero, interpret the key.  After the key has been
; interpreted, parse it: if it's a NAK (Control-U) clear the cmd_in_ptr;
; if it's a NULL (EOF), null terminate the command buffer and set the
; kybd_cmd_flag bit; otherwise just shove the character in the command buffe
; update the cmd_in_ptr (adjust it for overflow if necessary) and continue
; scanning.
;****************************************************************************

Pseudocode:

scan_kybd:
           save regs;
           row_number = (~kybd_rd_port) AND $F0;
           if (row_number = 0)
               then call no_key_pressed;
           bounce_count = bounce_count -1;
           if (bounce_count <= 0)
               then do;
                   if ((i_o_flags2 AND #key_parsed_flag) = 0)
                       then do;
                           i_o_flags2 = i_o_flags2 OR #key_parsed_flag;
                           j = key_intrp(row_number);
                           if (j < 0)
                               then do;
                                   if (j = $81)
                                       then column_number = $01;
                                   call no_key_pressed();
                               end;
                           else if (j = 0)
                               then do;
                                   command_buffer[cmd_in_ptr] = j;
                                   cmd_in_ptr = j;
                                   i_o_flags = i_o_flags OR #kybd_cmd_flag;
```

```
                              end;
                          else if (j = $15)
                              then cmd_in_ptr = 0;
                          else do;
                              command_buffer[cmd_in_ptr] = j;
                              cmd_in_ptr = cmd_in_ptr + 1;
                              if (cmd_in_ptr = $20)
                                  then do;
                                      call display_huh;
                                      cmd_in_ptr = 0;
                                  end;
                          end;
                      end;
              end;
          unsave regs;
      end scan_kybd;

no_key_pressed:
          i_o_flags2 = i_o_flags AND ~#key_parsed_flag;
          bounce_count = max_bounce_count;
          column_number >> 1;
          if (column_number = 0)
              then column_number = $08;
          if (alarm_flags < 0)
              then do;
                  kybd_wr_port = ((~column_number) OR ($20 AND write_port_
                                                  OR #bell_on;
                  write_portb = ((~column_number) OR ($20 & write_port_b))
                                                  OR #bell_on;
              end;
              else do;
                  kybd_wr_port = ((~column_number) OR ($20 & write_port_b)
                  write_portb = ((~column_number) OR ($20 & write_port_b))
          unsave regs;
      end no_key_pressed;

;************************************************************************
; The following routine is called with a (hopefully) non-zero value in t
; accumulator which represents the sensed keyboard row mask (in bits <7-
; This row number and the corresponding column number are mapped into ar
; index into an array of ASCII values.  The appropriate ASCII value for
; the key pressed is returned in the accumulator.  If a zero row mask wa
; provide to this routine, $80 is returned.  If the column number was ou
; of its allowable range $81 is returned.  If more than one key is press
; (row mask has more than one bit set), the "first key" (first bit set)
; is the one that is mapped to an ASCII value. The "#" key is mapped to
; the ASCII value $00 so that it can easily be recognized as the EOF
; character.
;
; If the mapped ASCII value is a number (0-9) then the front panel displ
; is updated by scrolling the new digit in from the right (least signifi
; digit).  If the "*" key was pressed indicating an operator entry error
; we get real fancy and clear the display.
;************************************************************************

Pseudocode:
```

```
ey_intrp:
      save regs;
      if ((j = row_map[value >> 4]) >= 0);     /* j = $80 */
          then do;
              j = j + col_map[column_number];
              if ((j = keybrd_map[j]) > 0);    /* j = $81 or j = 0 */
                  then do;
                      if (j = $15)
                          then do;
                              do i = 3 to 0 step -1;
                                  display_word[i] = #$80;
                                  display_word[4] = 0;
                                  call display_hndlr;
                          end;
                          else if (cmd_in_ptr = 0)
                              then do;
                                  do i = 2 to 0 step -1;
                                      display_word[i] = #$80;
                                      multimeter_flag = 0;
                                  end;
                                  k = keybrd_hex[y];
                                  do i = 3 to 0 step -1;
                                      display_word[i] = display_word[i-1];
                                      display_word[0] = k;
                                      display_word[4] = 0;
                                      call display_hndlr;
                                  end;
                              end;
                  end;
          end;
      unsave regs;
  end key_intrp;
```

===

KEY_INTRP TABLES

===

The following array provides the mapping for converting the sensed keyboard row number into the next level array index. This array also performs the function of finding the "first set bit" (in case multiple keys are pressed) and returns an error value ($80) for no key pressed at all.

```
ow_map:
      .byte $80,$03,$02,$02
      .byte $01,$01,$01,$01
      .byte $00,$00,$00,$00
      .byte $00,$00,$00,$00
```

The following array provides the mapping for converting the "column scanned mask into an array index that can be added to the index from the row_map in order to find out which one key has been pressed. For index values into this array that have none or more than one bit set, a "next-level" index is returned that will map into an invalid ASCII key value (i.e. the key will be $81).

```
col_map:
        .byte $10,$0C,$08,$10
        .byte $04,$10,$10,$10
        .byte $00,$10,$10,$10
        .byte $10,$10,$10,$10
;
; This array is the one that actually produces an ASCII value correspon
; to the key pressed.  It also produces error indications for malformed
; column scan values.
;
keybrd_map:
        .byte $44,$43,$42,$41   ; "D","C","B","A"
        .byte $00,$39,$36,$33   ; "#","9","6","3"
        .byte $30,$38,$35,$32   ; "0","8","5","2"
        .byte $15,$37,$34,$31   ; "*","7","4","1"
        .byte $81,$81,$81,$81   ; Column scan error return
;
; This array is analagous to the one above except that it contains hexa
; values of the keys pressed to speed up the display update function.  S
; having to convert ASCII back to HEX.
;
keybrd_hex:
        .byte $0D,$0C,$0B,$0A
        .byte $00,$09,$06,$03
        .byte $00,$08,$05,$02
        .byte $00,$07,$04,$01

;*************************************************************************
; This routine tests for rcve and xmit interrupt condx from the serial i
; port, services all valid ones and aborts if none exist.  This routine
; not a true subroutine, in that, it lies "in-line" as part of the overa
; IRQ service routine which also includes break.
;
;       It supports such user amenities as:
;
;               1. ^H, del, backspace - delete last character
;
;               2. ^U - "flush" command_buffer
;
;               3. adds a LF whenever a CR is received
;
;               4. ^Z - return to cmd_intrp
;
;               5. "-" - used to "backup" when using the open memory
;                       command in the monitor
;
; Upon receipt of a CR, a LF is sent out, and the serial_cmd_flag is set
; All spaces are deleted as they are input.
;
; The command buffer is assumed to be 32 bytes long and linear, i.e. the
; pointers must be zeroed when the data has been used by the called func
;*************************************************************************

Pseudocode:

serial_port:
```

```
        save regs;
        uart_status = pia_portb;
        if (uart_status AND bit 7 <> 0)
            then do;
                cntrl_z_flag = 0;
                char = uart;
                if (char < $20)
                    then do;
                        if (char = $08 OR char = $7F)
                            then do;
                                call msg_hndlr('^H ^H');
                                cmd_in_ptr = cmd_in_ptr - 1;
                            end;
                        else if (char = $0A OR char = $0D)
                            then do;
                                call putchar('CR');
                                call putchar('LF');
                                command_buffer[cmd_in_ptr] = 0;
                                cmd_in_ptr = 0;
                                serial_cmd_flag = 1;
                            end;
                            else do;
                                call putchar('^');
                                char = char + $40       ; make it printable
                                call putchar(char);
                                if (char = $15)
                                    then do;
                                        cmd_in_ptr = 0;
                                        call putchar('CR');
                                        call putchar('LF');
                                        call putchar('*');
                                    end;
                                if (char = $1A)
                                    then do;
                                        cntrl_z_flag = 1;
                                        call putchar('CR');
                                        call putchar('LF');
                                    end;
                            end;
                   end;
                   else do;
                        call putchar(char);     /* echo the character */
                        command_buffer[cmd_in_ptr] = char;
                        cmd_in_ptr = cmd_in_ptr + 1;
                        if (multimeter_flag = 1)
                            then multimeter_addr = @display_default - 1;
                        if (char = $2D) then look_bkward = 1;
                   end;
                end;
        if (uart_status AND bit 6 <> 0)
            then do;
                if (byte_count <> 0)
                    then do;
                        usrt = output_buffer[put_out]
                        put_out = put_out + 1;
                        decr byte_count;
```

```
                    end;
                    else disable xmit ints;
                end;
            unsave regs;
        end serial_port;

;**********************************************************************
; This routine services the BREAK instruction trap and the IRQ interrup
; It determines which of the two it was that vectored the CPU to this
; routine.  If it was a breakpoint, it continues, if an IRQ occurred in
; control is passed to serial_port for servicing.
;**********************************************************************

Pseudocode:

break:
        if (B_flag <> 1) then go to serial_port;
            else do;
                x_reg_stor = contents of X reg;
                accum_stor = contents of accumulator;
                flags_stor = psw popped off stack;
                y_reg_stor = contents of Y reg;
                pop "useless" PC off stack;
                pc_stor = brkpt_addr;
                sp_stor = contents of the stack pointer;
                call msg_hndlr ('PC = ');
                call putbyte (msbyte of pc_stor);
                call putbyte (lsbyte of pc_stor);
                call msg_hndlr ('  A = ');
                call putbyte (accum_stor);
                call msg_hndlr ('  Y = ');
                call putbyte (y_reg_stor);
                call msg_hndlr ('  X = ');
                call putbyte (x_reg_stor);
                call msg_hndlr ('  SP = ');
                call putbyte (sp_stor);
                call msg_hndlr ('  P = ');
                call putbyte (flags_stor);
                call msg_hndlr ('CRLF');
                call rem_brkpt;
                enable interrupts;
                go to run_task_master;
            end;
    end break;

;**********************************************************************
; This routine begins execution at the current user PC location or at t
; location specified in the command buffer, if any.  A, X, Y, P, and S
; always loaded from their respective storage locations.
;**********************************************************************

Pseudocode:

go:
        cmd_out_ptr = 1;
        if (command_buffer[1] <> 0)
```

```
          then do;
            msbyte of pc_stor = getbyte();
            if (ascii_hex_flag = 0)
              then do;
                cmd_out_ptr = 0;
                call put_huh;
                go to end go;
              end;
            lsbyte of pc_stor = getbyte();
            if (ascii_hex_flag = 0)
              then do;
                cmd_out_ptr = 0;
                call put_huh;
                go to end go;
              end;
          end;
          unsave 8 addresses and regs saved by cmd_intrp;
          cmd_out_ptr = 0;
          stack pointer = sp_stor;
          X reg = x_reg_stor;
          Y reg = y_reg_stor;
          accumulator = msbyte of pc_stor;
          push accumulator;
          accumulator = lsbyte of pc_stor;
          push accumulator;
          accumulator = flags_stor;
          push accumulator;
          accumulator = accum_stor;
          rti
    end go;

;****************************************************************************
; This function places a "BRK" opcode ($00) at the address specified in the
; command buffer.  Only one breakpoint is supported.  The replaced opcode is
; stored in rep_opcode.  The breakpoint address is stored in brkpt_addr.
;****************************************************************************

Pseudocode:

set_brkpt:
       save regs;
       cmd_out_ptr = 1;
       msbyte of brkpt_addr = getbyte();
       if (ascii_hex_flag = 0)
         then call put_huh;
         else do;
           lsbyte of brkpt_addr = getbyte();
           if (ascii_hex_flag = 0)
             then call put_huh;
             else do;
               rep_opcode = mem[brkpt_addr];
               mem[brkpt_addr] = 0;
             end;
         end;
       unsave reg;
    end set_brkpt;
```

```
;*********************************************************************
; This routine returns the opcode that was replaced with the "BRK".  It
; be called by the user and also the break subroutine.
;*********************************************************************

Pseudocode:

rem_brkpt:
        save regs;
        mem[brkpt_addr] = rep_opcode;
        unsave regs;
    end rem_brkpt:

;*********************************************************************
; This routine set the dump_state_flag to initiate dump_state from the
; serial port or the front panel keypad.
;*********************************************************************

Pseudocode:

set_up_dump_kybd:
set_up_dump:
        save regs;
        dump_state_flag = 1;
        unsave regs;
    end set_up_dump;

;*********************************************************************
; This routine, called from the front panel keypad, clears the valid_pas
; flag.
;*********************************************************************

Pseudocode:

kill_password:
        save regs;
        valid_password = 0;
        unsave regs;
    end kill_password;

;*********************************************************************
; This routine is called via the serial port and the hex keypad and "tog
; the state of the run_bit so that the system alternates between RUN and
; TEST/CALIBRATE modes.
;*********************************************************************

Pseudocode:

toggle_run_bit:
        save regs;
        compute_flags = compute_flags XOR run_flag;
        unsave regs;
    end toggle_run_bit;

;*********************************************************************
```

```
; This routine is called whenever the dump_state_flag is set to dump out the
; "state of the machine" to the serial port.  The output is composed of the
; contents of the dump_array with each element converted into the appropriate
; units.  The format for the output is as follows:
;
;        TIME: hh:mm
;        E00 = xxx.x VOLTS       I00 = xxx.x AMPS
;        E01 = xx.xx AMPS        I01 = xx.xx AMPS
;        E02 = xx.xx AMPS        I02 = xx.xx AMPS
;        E03 = xx.xx AMPS        I03 = xx.xx AMPS
;        E04 = xx.xx AMPS        I04 = xx.xx AMPS
;        E05 = xx.xx AMPS        I15 = xx.xx AMPS
;        E06 = xxx.x VOLTS       I06 = xxx.x VOLTS
;        E07 = xxx.x VOLTS       I07 = xxx.x VOLTS
;        E08 = xxx.x VOLTS       I08 = xxx.x VOLTS
;        E09 = xxx.x VOLTS       I09 = xxx.x VOLTS
;        E10 = xxx.x VOLTS       I10 = xxx.x VOLTS
;        E31 = xxx.x VOLTS       I31 = xxx.x VOLTS
;        E32 = xxx.x VOLTS       I32 = xxx.x VOLTS
;        E33 = xxx.x VOLTS       I33 = xxx.x AMPS
;        E34 = xxx.x AMPS        I34 = xxx.x AMPS
;        E35 = xxx.x AMPS        I35 = xxx.x AMPS
;        D36 = xxx.x DEG C       D37 = xxx.x DEG C
;        E38 = x.xxx VOLTS       E39 = x.xxx VOLTS
;        D00 = xxx%  D40 = xxx%  D41 = xxx%  D42 = xxx%
;        S1-1 S2-0               L1-1 L2-0 L3-1 L4-0 L5-1 L6-0
;        A1-1 A2-1 A3-1 A4-1 A5-1 A6-1 B1-1 B2-1 B3-1 B4-1 B5-1 B6-1
;        TOT CHGR I  = xxx.x AMPS        NOTE: 1=ON, 0=OFF
;
; Each line is formated and placed into the serial output buffer.  The
; routine is called again when the buffer is empty.  When all of the data
; has been transmitted, the dump_state_flag and valid_password are both
; cleared.  In addition, the address of the proper display default routine
; is loaded into multimeter_addr, the channel data (in ascii), if needed,
; is loaded into multimeter_data and finally, the multimeter_flag is set.
; Every one second after this, the indicated display routine is re-initiated.
;***************************************************************************

seudocode:

ump_state:
        save regs;
        if (cntrl_z_flag = 1) then goto _cntrl_z_crash;
        if (dump_channel = 0) then call put_time;
        dump_count = 0;
        do while ((dump_count <= 1) AND (dump_channel < soc_channel_num));
            accum = label_list[dump_channel];
            call putchar(accum);
            mem_byte = user_ch_table[dump_channel];
            call putbyte(mem_byte);
            call msg_hndlr(' = ');
            conv_dump_addr = conv_dump_table[dump_channel*2];
            call convert_and_output routine;
            dump_channel = dump_channel + 1;
        end;
        if (dump_channel = soc_channel_num)
```

```
            then do;
                call dump_soc;
                call dump_sw_states;
                call dump_arrays;
                call dump_total_I;
                call msg_hndlr('        NOTE: 1=ON, 0=OFF');
    cntrl_z_crash:
                dump_channel = 0;
                dump_state_flag = 0;
                valid_password = 0;
                multimeter_flag = 1;
                multimeter_addr = @display_default - 1;
                call putchar('CR');
                call putchar('LF');
                call putchar('*');
            end;
            else do;
                call putchar('CR');
                call putchar('LF');
            end;
        unsave regs;
    end dump_state;

;==================================================================================
;                              DUMP_STATE LOOKUP TABLES
;==================================================================================

label_list: .ascii      "EIEIEIEIEIEIEIEIEIEIEIEIEIEIEIEIDDEE"

user_ch_table:
        .byte   $00, $00, $01, $01, $02, $02, $03, $03, $04, $04, $05, $
        .byte   $06, $06, $07, $07, $08, $08, $09, $09, $10, $10, $31, $
        .byte   $32, $32, $33, $33, $34, $34, $35, $35, $36, $37, $38, $

conv_dump_table:

        NOTE: The -1 in each of the following is due to a peculiarity of
        6502.  Namely that the address stored on the stack during a jsr
        1 LESS THAN the return address.  So in order to use the stack ar
        RTS to impliment an indirect JSR, the address stored in the tabl
        must conform to this rather "arbitrary" condition.

        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
```

```
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_volts - 1
        .word   dump_amps - 1
        .word   dump_temp - 1
        .word   dump_temp - 1
        .word   dump_volts - 1
        .word   dump_volts - 1

;*************************************************************************
; This routine fill s specified region in RAM with a specified byte of data
; The starting address, the ending address and the datum are contained in the
; command buffer.
;
;       WATCH USING THIS ROUTINE BELOW LOCATION $320, THE RESULTS MAY BE
;       UNDESIREABLE BECAUSE IT MAY OVERWRITE ZERO PAGE, THE STACK OR THE
;       I/O BUFFERS.
;
;*************************************************************************

Pseudocode:

Fill:
        save regs;
        cmd_out_ptr = 1;
        msbyte of start_addr = getbyte();
        if (ascii_hex_flag = 0) then call put_huh;
          else do;
            lsbyte of start_addr = getbyte();
            if (ascii_hex_flag = 0) then call put_huh;
              else do;
                msbyte of ending_addr = getbyte();
                if (ascii_hex_flag = 0) then call put_huh;
                  else do;
                    lsbyte of ending_addr = getbyte();
                    if (ascii_hex_flag = 0) then call put_huh;
                      else do;
                        mem_byte = getbyte();
                        if (ascii_hex_flag = 0) then call put_huh;
                          else do;
                            if (ending_addr - start_addr < 0)
                              then call put_huh;
```

```
                                   else do;
                                     do while (ending_addr - start_addr >= 0)
                                       mem[start_addr] = mem_byte;
                                       incr start_addr;
                                     end;
                                   end;
                               end;
                           end;
                       end;
                   end;
               end;
             cmd_out_ptr = 0;
             unsave regs;
         end fill;

;*****************************************************************************
; This routine handles the storage function of data which is downloaded
; another computer via a serial link.  The selected download format is t
; used with the DATA I/O PROM Programmer, i.e.:
;
;       ^B$A<address>, XX XX XX XX XX...^CS<checksum>,zzzzzzzzzzzz
;
;                       where 1. XX = ascii encoded hex digits
;                             2. address and checksum are represented ir
;                                hex digits
;                             3. the acceptable delimiters are space, CF
;                                LF.
;
; In order to maintain as much universality as possible, it is desireabl
; make the routine independent of interrupt service routines which may u
; buffers of undefined form.  Hence, IRQ's are disabled for the duratior
; the download.
;*****************************************************************************

Pseudocode:

dwnld:
        turn off interrupts;    /* this routine uses polled ACIA only */
        checksum = 0;
        first_nyb_flag = 0;
        error_flag = 0;
        char = p_getchar();
        do while (char <> $02);         /* look for ^B */
                char = p_getchar();
        end;
        do while ((char <> $03) AND (error_flag = 0));  /* look for ^C *
            if (char = '$')
                then do;
                    char = p_getchar();
                    do while (char <> 'A');
                        char = p_getchar();
                    end;
                    mem_addr = p_getword();     /* get the address */
                    offset = 0;
                    call p_getcomma();  /* get the comma */
                end;
```

```
                else do;
                    if (char <> $0A) OR (char <> $0D) OR (char <> ' ')
                        then do;
                            if (first_nyb_flag = 0)
                                then do;
                                    nybble_1 = a_to_h(char);
                                    if (nybble_1 = $FF) then error_flag = $F
                                    first_nyb_flag = $FF
                                end;
                                else do;
                                    nybble_2 = a_to_h(char);
                                    if (nybble_2 = $FF) then error_flag = $F
                                    byte = nybble_1:nybble_2;
                                    mem[mem_addr] = byte;
                                    checksum = checksum + byte;
                                    first_nyb_flag = 0;
                                    mem_addr = mem_addr + 1;
                                end;
                        end;
                end;
                    char = p_getchar();
        end;
        if (error_flag <> 0)
            then call p_msg_hndlr('bad_data')
            else do;
                char = p_getchar();
                do while (char <> 'S'); /* look for the S */
                    char = p_getchar();
                end;
                xmt_chk_sum = p_getword();
                if (checksum <> xmt_chk_sum)
                    then call p_msg_hndlr('BAD RUN');
                    else call p_msg_hndlr('GOOD RUN');
            end;
    end download;

;****************************************************************************
; This routine obtains the current version number of this software, and
; transmits it to the serial port.
;       IMPORTANT: the assumption is that the version number is always
;                  two BCD digits of the form M.N
;****************************************************************************

Pseudocode:

put_version:
        save regs;
        accum = (version_num AND $F0) >> 4;
        call putchar(accum);
        call putchar('.');
        accum = version_num AND $0F;
        call putchar(accum);
        unsave regs;
    end put_version;

;****************************************************************************
```

```
; This routine obtains the current version number of this software, and
; displays it, properly formatted on the hexpad display panel.
;
;       IMPORTANT: the assumption is that the version number is always
;                  two BCD digits of the form M.N
;************************************************************************

Pseudocode:

out_version:
        save regs;
        display_word[4] = 3;
        display_word[3] = (version_num AND $F0) >> 4;
        display_word[2] = version_num AND $0F;
        display_word[1] = $80;
        display_word[0] = $80;
        call display_hndlr;
        unsave regs;
    end out_version;

;************************************************************************
; This is a 8 x 8 multiply similar to mp_mult except used only for singl
; precision multiplication.  The multiplier is loaded into sp_mult1, the
; multiplicand in sp_mult2 and the 16 bit product is returned in sp_proc
;************************************************************************

Pseudocode:

sp_mult:
        save regs;
        msbyte of sp_mult2 = 0;
        sp_product = 0;
        do i = 0 to 7;
            sp_mult1 = sp_mult1 >> 1;
            if (carry = 1) then sp_product = sp_product + sp_mult2;
            sp_mult2 = sp_mult * 2;
        end;
        unsave regs;
 end sp_mult;

;************************************************************************
; This routine sets the duty cycle of the selected PWM timer to the desi
; value.  The value input by the user is given as a %-age duty cycle, wi
; the input %-age being the "time high".  The value for selecting the ti
; can be either "1" or "2".  If invalid entries are made, error messages
; are sent to the terminal.
;************************************************************************

Pseudocode:

set_duty_cycle:
        save regs;
        if ((command_buffer[1] < #$31) OR (command_buffer[1] > #$32))
            then call msg_hndlr('DIGIT NOT 1 OR 2');
            else do;
                temp = command_buffer[1] - #$31;
```

```
              cmd_out_ptr = 2;
              x = getbyte();
              if ((i_o_flags & #ascii_hex_flag) = 0)
                  then call msg_hndlr('DIGIT OUT OF RANGE 0-F');
                  else do;
                      y = d_to_h(x) * 2;
                      if ((compute_flags & #over_10_flag) != 0)
                          then call msg_hndlr('DIGIT OUT OF RANGE 0-9');
                          else do;
                              if (y > #$AE)
                                  then call msg_hndlr('DUTY CYCLE VALUE OU
                                              OF RANGE 0-87%');
                                  else do;
                                      timer_csr = #$E0;
                                      if (temp == 0)
                                        then do;
                                            timer_csr = #$C4;
                                            timer_csr = #$0B;
                                            timer_data = duty_low_tbl[y];
                                            timer_data = duty_low_tbl[y+1
                                            timer_data = duty_hi_tbl[y];
                                            pwm_value = duty_hi_tbl[y];
                                            timer_data = duty_hi_tbl[y+1]
                                            timer_csr = #$E3;
                                            timer_csr = #$24;
                                        end;
                                        else do;
                                            timer_csr = #$C8;
                                            timer_csr = #$0C;
                                            timer_data = duty_low_tbl[y];
                                            timer_data = duty_low_tbl[y+1
                                            timer_data = duty_hi_tbl[y];
                                            timer_data = duty_hi_tbl[y+1]
                                            timer_csr = #$E4;
                                            timer_csr = #$28;
                                       end;
        unsave regs;
    end set_duty_cycle;

===========================================================================
               PERCENT DUTY CYCLE TO PWM COUNT TABLE
===========================================================================

uty_lo_tbl:
       .byte   $C6,$00,$C6,$00,$C4,$00,$C2,$00
       .byte   $C0,$00,$BE,$00,$BC,$00,$BA,$00
       .byte   $B8,$00,$B6,$00,$B4,$00,$B2,$00
       .byte   $B0,$00,$AE,$00,$AC,$00,$AA,$00
       .byte   $A8,$00,$A6,$00,$A4,$00,$A2,$00
       .byte   $A0,$00,$9E,$00,$9C,$00,$9A,$00
       .byte   $98,$00,$96,$00,$94,$00,$92,$00
       .byte   $90,$00,$8E,$00,$8C,$00,$8A,$00
       .byte   $88,$00,$86,$00,$84,$00,$82,$00
       .byte   $80,$00,$7E,$00,$7C,$00,$7A,$00
       .byte   $78,$00,$76,$00,$74,$00,$72,$00
       .byte   $70,$00,$6E,$00,$6C,$00,$6A,$00
```

```
        .byte   $68,$00,$66,$00,$64,$00,$62,$00
        .byte   $60,$00,$5E,$00,$5C,$00,$5A,$00
        .byte   $58,$00,$56,$00,$54,$00,$52,$00
        .byte   $50,$00,$4E,$00,$4C,$00,$4A,$00
        .byte   $48,$00,$46,$00,$44,$00,$42,$00
        .byte   $40,$00,$3E,$00,$3C,$00,$3A,$00
        .byte   $38,$00,$36,$00,$34,$00,$32,$00
        .byte   $30,$00,$2E,$00,$2C,$00,$2A,$00
        .byte   $28,$00,$26,$00,$24,$00,$22,$00
        .byte   $20,$00,$1E,$00,$1C,$00,$1A,$00

duty_hi_tbl:
        .byte   $02,$00,$02,$00,$04,$00,$06,$00
        .byte   $08,$00,$0A,$00,$0C,$00,$0E,$00
        .byte   $10,$00,$12,$00,$14,$00,$16,$00
        .byte   $18,$00,$1A,$00,$1C,$00,$1E,$00
        .byte   $20,$00,$22,$00,$24,$00,$26,$00
        .byte   $28,$00,$2A,$00,$2C,$00,$2E,$00
        .byte   $30,$00,$32,$00,$34,$00,$36,$00
        .byte   $38,$00,$3A,$00,$3C,$00,$3E,$00
        .byte   $40,$00,$42,$00,$44,$00,$46,$00
        .byte   $48,$00,$4A,$00,$4C,$00,$4E,$00
        .byte   $50,$00,$52,$00,$54,$00,$56,$00
        .byte   $58,$00,$5A,$00,$5C,$00,$5E,$00
        .byte   $60,$00,$62,$00,$64,$00,$66,$00
        .byte   $68,$00,$6A,$00,$6C,$00,$6E,$00
        .byte   $70,$00,$72,$00,$74,$00,$76,$00
        .byte   $78,$00,$7A,$00,$7C,$00,$7E,$00
        .byte   $80,$00,$82,$00,$84,$00,$86,$00
        .byte   $88,$00,$8A,$00,$8C,$00,$8E,$00
        .byte   $90,$00,$92,$00,$94,$00,$96,$00
        .byte   $98,$00,$9A,$00,$9C,$00,$9E,$00
        .byte   $A0,$00,$A2,$00,$A4,$00,$A6,$00
        .byte   $A8,$00,$AA,$00,$AC,$00,$AE,$00

;*************************************************************************
; This routine, called from either the terminal or the front panel keypa
; allows the user to set and clear individual bits that control various
; "devices".  These devices are divided into various "classes" which de
; which section of code handles the device specific functions.  The "dev
; which can be controlled include the audio buzzer, the warning LEDs, th
; user's load requests, the PWM control ports, and the load current over
; trip control bits.
;
;       device 0                audible alarm
;       device 1-6              user load requests 1 thru 6
;       device 7-11             overload trip resets for loads 1 thru 5
;       device 12-17            PWM buffer #1 controls 1 thru 6
;       device 18-23            PWM buffer #2 controls 1 thru 6
;       device 24-25            Yellow, and Red LEDs
;
; If a digit out of the range of 0 to 9 is entered for either digit of t
; bit number or out of the range 0 to 1 for the desired state (1=ON), or
; the bit number exceeds 25, then an appropriate error message is sent t
; controlling I/O device (terminal or keypad).  In addition, if the user
; attempts to set a bit in the range 6 - 10, the equivalent of "tripping
```

```
; a load's circuit breaker, a warning message is displayed and no action is
; taken.
;***************************************************************************

Pseudocode:

set_bit:
        save regs;
        if((command_buffer[1] < $30) OR (command_buffer[1] > $39))
           then do;
              if (command_buffer[0] = 'D')
                 then call display_error(#$04);
                 else call msg_hndlr('DIGIT OUT OF RANGE 0-9');
           end;
           else do;
              sb_temp = command_buffer[1] << 4;
              if((command_buffer[2] < $30) OR (command_buffer[2] > $39))
                 then do;
                    if (command_buffer[0] = 'D')
                       then call display_error(#$04);
                       else call msg_hndlr('DIGIT OUT OF RANGE 0-9');
                 end;
                 else do;
                    sb_temp = sb_temp OR (command_buffer[2] AND #$0F);
                    if ((command_buffer[3] < $30) OR
                                        (command_buffer[3] > $31))
                       then do;
                          if (command_buffer[0] = 'D')
                             then call display_error(#$07);
                             else call msg_hndlr('DIGIT NOT A '1' OR '2'');
                       end;
                       else do;
                          if(command_buffer[3] = $30)
                             then cflag = 0;
                             else cflag = 1;
                          if(sb_temp >= max_num_devs)
                                              /* max_num_devs = #$26 */
                            then do;
                               if(command_buffer[0] = 'D')
                                  then call display_error(#$09);
                                  else call msg_hndlr('INVALID DEVICE NUMBER
                            end;
                            else do;
                               if(sb_temp >= $20)
                                  then sb_temp = sb_temp - #$0C;
                                  else if(sb_temp >= $10)
                                     then do;
                                        sb_temp = sb_temp - #$06;
                                        sb_temp = sb_temp << 1;
                                        i = sb_msk_tbl[sb_temp];
                                        j = sb_msk_tbl+1[sb_temp];
        switch (j);
                case 0:
                    if (cflag = 1)
                        then alarm_flags = alarm_flags OR i;
                        else alarm_flags = alarm_flags AND ~i;
```

```
                    break;
                case 1:
                    if(cflag = 1)
                        then user_ld_req = user_ld_req OR i;
                        else user_ld_req = user_ld_req AND ~i;
                    break;
                case 2:
                    if (cflag = 1)
                        then do;
                           ovrld_trip = ovrld_trip AND ~i;
                           ovrld_cnt = ovrld_cnt_max(sb_temp - 7);
                           i = #enable_bell;
                           alarm_flags = alarm_flags AND ~i;
                        end;
                        else do;
                           if (command_buffer[0] = 'D')
                              then call display_error(#$08);
                              else call msg_hndlr('CAN'T TRIP OVERLOAD
                                                          PROTECTOR');
                        end;
                    break;
                case 3:
                    if(cflag = 1)
                        then call cntrl_pwm_output(i OR #$80);
                        else call cntrl_pwm_output(i);
                    break;
                case 4:
                    if(cflag = 1)
                        then stow_leds = stow_leds OR i;
                        else do;
                            stow_leds = stow_leds AND ~i;
                            led_out_latch = stow_leds;
                        end;
                    break;
                         end;
                     end;
                  end;
           end;
       unsave regs;
   end set_bit;

;========================================================================
;                              SET BIT MASK TABLE
;========================================================================

sb_msk_tbl:
        .byte   $80,$00          ; audio buzzer, Class $0
        .byte   $08,$01          ; User load requests 1 - 6, Class 1
        .byte   $10,$01
        .byte   $20,$01
        .byte   $40,$01
        .byte   $80,$01
        .byte   $04,$01
        .byte   $08,$02          ; Overload trip controls 1- 5, Class 2
        .byte   $10,$02
        .byte   $20,$02
```

```
        .byte   $40,$02
        .byte   $80,$02
        .byte   $00,$03         ; PWM Control arrays 1 & 2 (6 outputs each)
        .byte   $01,$03         ; Class 3
        .byte   $02,$03
        .byte   $03,$03
        .byte   $04,$03
        .byte   $05,$03
        .byte   $06,$03
        .byte   $07,$03
        .byte   $08,$03
        .byte   $09,$03
        .byte   $0A,$03
        .byte   $0B,$03
        .byte   $01,$04         ; LED controls (Yellow,Red) Class 4
        .byte   $02,$04

;****************************************************************************
; This routine takes as its input the contents of the accumulator and
; sets or clears the selected PWM control output bit.  The contents of
; the accumulator are encoded as follows: Bit 7 is a "1" if the PWM control
; bit is to be set, a "0" if the PWM control bit is to be cleared.  Bits 0-3
; are the number (or "address" if you will) of the bit to be set/cleared.
; Values between $00 and $05 are in PWM control array 1, and values between
; $06 and $0B are in PWM control array 2.  The bit number is used as an
; index into the array pwm_msk_tbl whose entries are the bit pattern to
; be used in turning on or off the PWM control.  In addition, if bit 7 of th
; pwm_tbl_msk array entry is on then the mask is for PWM control array 2.
;****************************************************************************

Pseudocode:

cntrl_pwm_output:
        save regs;
        i = pwm_msk_tbl[value AND #$0F]
        if (i >= 0)
            then do;
                if (value < 0)
                    then do;
                        stow_pwm1 = stow_pwm1 OR i;
                        array_sel1 = stow_pwm1;
                    end;
                    else do;
                        stow_pwm1 = stow_pwm1 AND ~i;
                        array_sel1 = stow_pwm1;
                    end;
            end;
            else do;
                i = i AND #$7F;
                if (value < 0)
                    then do;
                        stow_pwm2 = stow_pwm2 OR i;
                        array_sel2 = stow_pwm2;
                    end;
                    else do;
                        stow_pwm2 = stow_pwm2 AND ~i;
```

```
                        array_sel2 = stow_pwm2;
                    end;
                end;
            unsave regs;
        end cntrl_pwm_output;

;==========================================================================
;                PULSE MASK TABLE FOR CNTRL_PWM_OUTPUT
;==========================================================================

pwm_msk_tbl:
        .byte $01,$02,$04,$08,$10,$20
        .byte $81,$82,$84,$88,$90,$A0

;**************************************************************************
; This routine converts the A/D reading in frez_temp into a "%-age coldr
; product, which is an indication of the state of the freezer load.
;**************************************************************************

Pseudocode:

find_prod:
        save regs;
        if (frez_temp[1] < 0)
            then i = 0;
            else do;
                i = frez_temp[1] >> 1;
                save flags;
                i = i >> 1;
                save flags;
                i = (frez_temp[0] >> 3);
                unsave flags;
                if (carry bit = 1)
                then i = i OR $40;
                unsave flags;
                if (carry bit = 1)
                then i = i OR $20;
                i = i - $50;
                if (i < 0)
                    then i = 0;
                    else if (i > $35)
                        then i = $35;
            end;
        prod = prod_pct_tbl[i];
        unsave regs;
    end find_prod;

;==========================================================================
;               PERCENTAGE OF PRODUCT TABLE FOR FIND-PROD
;==========================================================================

prod_pct_tbl:
        .byte  2,  4,  6,  8, 10, 12, 14, 16, 20, 22, 24, 26
        .byte 28, 30, 32, 34, 36, 40, 42, 44, 46, 48, 50, 52
        .byte 54, 56, 60, 62, 64, 66, 68, 70, 72, 74, 78, 80
        .byte 82, 84, 86, 88, 90, 92, 94, 98, 100, 102, 104, 106
```

```
        .byte 108, 110, 112, 116; 118, 120

;***************************************************************************
; This routine, called on a one second interval, tests to see if any of the
; string currents have fallen below a specified level, i.e. if the following
; expression is false for any of the strings, then the string_fault flag is
; set.
;
;  branch current(N) >= (total_chgr_I/number of strings) - str_fault_offset
;***************************************************************************

'suedocode:

:est_string_I:
        save regs;
        lsword of dividend = total_chgr_I;
        msword of dividend = 0;
        lsbyte of divisor = num_pwr_strings;
        msbyte of divisor = 0;
        quotient = mp_div();
        x = 0;
        quotient = quotient - str_fault_offset;
        do while ((x < num_pwr_strings) AND (branch1[ x ] >= quotient))
            x = x + 1;
        end;
        if (x = num_pwr_strings)
            then string_fault = 0;
            else string_fault = 1;
        unsave regs;
    end test_string_I;


 ***************************************************************************
  This routine handles the dumping of state_of_chg, corrected_state_of_charge
  equalization count and pwm_value for the dump state routine.
 ***************************************************************************

 seudocode:

 ump_soc:
        save regs;
        call putchar('D');
        call putbyte($00);
        call msg_hndlr(' = ');
        call put_soc;
        call putchar($20);
        call putchar($20);
        call putchar('D');
        call putchar($40);
        call msg_hndlr(' = ');
        call put_csoc;
        call putchar($20);
        call putchar($20);
        call putchar('D');
        call putchar($41);
        call msg_hndlr(' = ');
```

```
        call put_equal;
        call putchar($20);
        call putchar($20);
        call putchar('D');
        call putchar($42);
        call msg_hndlr(' = ');
        call put_pwm;
        call putchar('CR');
        call putchar('LF');
        unsave regs;
    end dump_soc;

;*************************************************************************
; This routine handles the dumping of the switch and load states for th
; dump_state routine.
;*************************************************************************

Pseudocode:

dump_sw_states:
        save regs;
        call putchar('S');
        call putchar('1');
        accum = alarm_30 AND inhibit_bit;
        call put_on;
        call putchar('S');
        call putchar('2');
        accum = alarm_30 AND motor_temp;
        call put_on;
        call putchar('L');
        call putchar('1');
        accum = stow_leds AND bit3;
        call put_off;
        call putchar('L');
        call putchar('2');
        accum = stow_leds AND bit4;
        call put_off;
        call putchar('L');
        call putchar('3');
        accum = stow_leds AND bit5;
        call put_off;
        call putchar('L');
        call putchar('4');
        accum = stow_leds AND bit6;
        call put_off;
        call putchar('L');
        call putchar('5');
        accum = stow_leds AND bit7;
        call put_off;
        call putchar('L');
        call putchar('6');
        accum = stow_leds AND bit2;
        call put_off;
        call putchar('CR');
        call putchar('LF');
        unsave regs;
```

```
   end dump_sw_states;

 put_on:
       call putchar('-');
       if (accum = 0)
           then call putchar($31);
           else call putchar($30);
       call putchar(' ');
    end put_on;

 put_off:
       call putchar('-');
       if (accum = 0)
           then call putchar($30);
           else call putchar($31);
       call putchar(' ');
    end put_off;
;***************************************************************************
; This routine dumps the state of the 12 array enable lines during dump_state
;***************************************************************************

'seudocode:

jump_arrays:
       save regs;
       temp = stow_pwm1;
       do i=31 to 36;
           call putchar('A');
           call putchar(i);
           call putchar('-');
           temp = temp >> 1
           if (carry = 0)
               then call putchar('0');
               else call putchar('1');
           call putchar(' ');
       end;
       temp = stow_pwm2;
       do i=31 to 36;
           call putchar('A');
           call putchar(i);
           call putchar('-');
           temp = temp >> 1
           if (carry = 0)
               then call putchar('0');
               else call putchar('1');
           call putchar(' ');
       end;
       call putchar('CR');
       call putchar('LF');
       unsave regs;
   end dump_arrays;
```

```
XXXXXX      XXXXX   XXXXX     XXXX   X     X
X           X    X  X    X   X    X  XX   XX
XXXXX       X    X  X    X   X    X  X XX X
X           XXXXX   XXXXX    X    X  X     X
X           X       X   X    X    X  X     X
X           X       X    X    XXXX   X     X
```

```
;*****************************************************************************
; This is the "main" control module.  This module invokes all of the other
; modules, either directly or indirectly.  The routines called directly by
; this module are listed below. The functions of the run_task_master are man
; but they are broken down into 4 regimes.
;
;       1. background regime - the most fundamental loop for the system to
;       be in if nothing is happening.
;
;       2. dataset_ready regime - entered only when the new dataset is
;       available for signal averaging.
;
;       3. 100 msec regime - within the dataset_ready regime.  It is within
;       this area that the signal averaging multimeter functions and max
;       power control are done.
;
;       4. 1 second regime - this area is devoted primarily with the
;       handling of the clock and machine state considerations.
;*****************************************************************************
```

Pseudocode:

```
run_task_master:
        do while (1);
            if (dsply_pend_flag = 1) then call display_digits;
            if (dataset_ready_flag = 1)
                then do;
                    strobe watchdog timer;
                    if (serial_cmd_flag OR kybd_cmd_flag = 1)
                        then call cmd_intrp;
                    if (pwm_m_flg = 1)
                        then do;
                            call abs_cnvt;
                            if (run_flag = 1));
                                then do;
                                    call battery_state_of_chg;
                                    if (max_pw_cntrl = 1)
                                        then call max_pwr_track;
                                    if ((dump_state_flag = 1)
                                                AND (byte_count = 0))
                                        then call dump_state;
                                end;
                            dataset_ready_flag = 0;
                            pwm_m_flg = 0;
                            mmf_update = mmf_update - 1;
                        end;
                    if (mmf_update = 0)
                        then do;
                            call multimeter_func;
                            mmf_update = 3;
                        end;
                end;
            if (one_sec_flag = 1)
```

```
                then do;
                    call test_string_I;
                    call find_time;
                    if (alarm_30 AND bit6 <> 0)
                        then do;
                            "reset" the 30 min alarm;
                            dump_state_flag = 1;
                        end;
                    if (run_flag = 1)
                        then do;
                            if (max_pw_cntrl = 0)
                                then call discrete_array_cntrl;
                            call correct_state_of_chg;
                            call deter_mach_state;
                        end;
                        else stow_leds=led_out_latch=stow_leds XOR #rled_
                    call chk_for_ovrld;
                    one_sec_flag = 0;
               end;
        end;
    end run_task_master;

;*************************************************************************
; This routine is awakened when the pwm_m_flg is set to indicate that
; it is time to update the control point.
;
;       Note: if (direction <> 0) then duty cycle incr's
;             if (direction = 0) then duty cycle decr's
;
;*************************************************************************

Pseudocode:

max_power_track:
        save regs;
        old_power = new_power;
        new_power = (abs_battery_V + $100) * abs_total_chgr_I;
        delta_power = new_power - old_power;
        if ((delta_power < 0) AND (dir_chg_cntr = 0))
            then do;
                direction = direction XOR 1;
                dir_chg_cntr = 2;
                step = small;
            end;
            else if (delta_power > old_power * 2^(-big))
                then step = large;
                else step = small;
        if (pwm_value <= min_pwm)
            then do;
                direction = 1;
                step = large;
            end;
        if ((pwm_value >= max_pwm) OR ((total_chgr_I > max_chgr_I)
                        OR (abs_battery_V > bat_V_limit)
                                OR (motor_temp = 1)
                                        OR (string_1_V < min_array_volts)
```

```
          then do;
               direction = 0;
               step = small;
          end;
     if (direction_bit = 0) then step = - step;
     pwm_value = pwm_value + step;
     if (pwm_value > max_pwm)
          then if (direction <>0)
               then pwm_value = max_pwm;
               else pwm_value = 1;
     if ((pwm_value < 1 count) OR (inhibit = 0))
          then do;
               pwm_value = 1;
               turn_on_flag = 0;
          end;
          else turn_on_flag = 1;
          do;
               disable ints;
               timer_csr = $0B
               timer_data = $C8 - pwm_value;    /* load downtime */
               timer_data = $00
               timer_data = pwm_value;          /* load uptime */
               timer_data = $00
               if (turn_on_flag = 1)
                    then array_sel1=array_sel2=stow_pwm1=stow_pwm2 = #$3F;
                    else array_sel1=array_sel2=stow_pwm1=stow_pwm2 = #$0;
                    end;
               enable ints;
          end;
     settle_time = $03;
     if (dir_chg_cntr <> 0)
          then dir_chg_cntr = dir_chg_cntr - 1;
     unsave regs;
  end max_power_track;
```

****

This routine looks through a table of commands for a match to the first character that it finds in the command_buffer. The next two locations in the table hold the jump address of the routine to handle the requirement. The corresponding routine is invoked. Since the data in the command buffer could be the password, this is checked first. Failing this test, the input is tested to see if it is one of the possible command words. If so, that task is initiated. If no command word is matched a "WHAT?" message is sent to the appropriate port.

If the msb of the table character is set, it indicates that the function is a multimeter display function. In this case, the ascii representation of the channel number in the command buffer is moved to multimeter_data, the display function starting address is retained in multimeter_addr and the multimeter_flag is set.

Upon completion of this task the following actions are taken:

1. control is returned to cmd_intrp

2. a prompt, "CR LF *", is sent to the serial port, using

```
;                msg_hndlr
;
;                3. clear serial_cmd_flag or the kybd_cmd_flag as appropr
;*****************************************************************************

Pseudocode:

cmd_intrp:
        save regs;
        cmd_out_ptr = 0;
        if (kybd_cmd_flag OR serial_cmd_flag = 1)
            then do;
                if (kybd_cmd_flag = 1)
                    then do;
                        index = 0;
                        do while ((command_buffer[index] = password[inde
                                        AND (index < 4));
                            index = index + 1;
                        end;
                        if (index = 4)
                            then do;
                                valid_password = 1;
                                call display_clr();
                            end;
                            else do;
                                call display_clr();
                                if ((command_buffer[1] < $3A))
                                    then do;
                                       if (valid_password = 0) then
                                                base_addr = _cmd_table1;
                                           else base_addr = _cmd_table3;
                                       call _find_func;
                                       kybd_cmd_flag = 0;
                                       cmd_out_ptr = 0;
                                       cmd_in_ptr = 0;
                                    end;
                                    else do;
                                       cmd_out_ptr = 1;
                                       if (valid_password = 0) then
                                                base_addr = _cmd_table2;
                                           else base_addr = _cmd_table4;
                                       call _find_func;
                                       kybd_cmd_flag = 0;
                                       cmd_out_ptr = 0;
                                       cmd_in_ptr = 0;
                                    end;
                            end;
                    end;
                    else do;
                        base_addr = _cmd_table5;
                        call _find_func;
                        call msg_hndlr(prompt);
                        serial_cmd_flag = 0;
                        cmd_out_ptr = 0;
                        cmd_in_ptr = 0;
                    end;
```

```
            end;
        unsave regs;
    end cmd_intrp;

        _find_func: procedure;
            index = 0;
            do while ((command_buffer[cmd_out_ptr] <>
                        ($7F AND base_addr[index])) AND
                                (base_addr[index] <> 0));
                index = index + 3;
            end;
            if (base_addr[index] AND $80 <> 0)
                then do;
                    multimeter_flag = 1;
                    multimeter_data = command_buffer[1];
                    multimeter_data+1 = command_buffer[2];
                    multimeter_addr = base_addr[index + 1];
                end;
            if (base_addr[index] = 0)
                then do;
                    if(kybd_cmd_flag = 1) then call display_huh;
                        else call put_huh;
                end;
                else do;
                    jump_addr = base_addr[index + 1];
                    call routine located @ jump_addr;
                end;
            end _find_func;

****************************************************************************
 This routine gathers N channels of data, sequentially, and stores the
 results in memory.  The routine set the mux channel number, waits for
 settling, starts the conversion, polls for conversion completion and then
 stores the result.

 Note: entry_num, next_seldom and pass_num are zeroed at init
****************************************************************************

seudocode:

dc_hndlr:
        save regs;
        temp = pia_csra;
        if (temp AND bit 6 <> 0)
            then call _knock_down;
        if (temp AND bit 7 <> 0)
          then do;
            decr pwm_mod_timer;
            if (pwm_mod_timer = 0)
               then do;
                  pwm_mod_timer = 25;
                  pwm_m_flg = 1;
               end;
            call scan_kybd;
            if (settle_time = 0)
               then do;
```

```
                if (dataset_ready_flag = 0)
                   then do;

                              "OFTEN READS"

                       next_often = 0;
                       do while next_often <= num_of_often_reads;
                           select the often_read mux channel;
                           adc_lo = 0;                       to start convers
                           if (next_often = 0)
                              else do;
                                 rearrange adc_value;
                                 if (adc_value < 0) then remove sign b
                                              complement the value;
                                 often_read[entry_num,(next_often-1)*32
                                                  adc_value;
                              end;
                           do while (adc_lo AND adc_not_busy = 0);
                              end;              poll for conversion compl
                           adc_value = adc_hi:adc_lo;
                           next_often = next_often + 1;
                       end;
                       rearrange adc_value;
                       often_read[entry_num,(next_often-1)*32] = adc_va

                              "SELDOM READS"

                       if (next_seldom > num_of_seldom_reads)
                          then next_seldom = 0;
                          else next_seldom = next_seldom + 1;
                       select seldom_read mux channel;
                       adc_lo = 0;                  to start conversion
                       if (pass_num >= max_pass_num)
                          then do;
                             dataset_ready_flag = 1;
                             pass_num = 0;
                             entry_num = 0;
                          end;
                          else do;
                             pass_num = pass_num + 1;
                             entry_num = entry_num + 1;
                          end;
                       do while (adc_lo AND adc_not_busy = 0);
                          end;                      poll for conversion comp
                       seldom_read[next_seldom] = rearranged adc;
                   end;
            end;
            else settle_time = settle_time - 1;
                              this is the settle_time that is set
                              when the PWM value has been changed.
                              A value of 5 for settle_time will
                              suppress any current measurements for
                              20 msec
         if ((one_sec_flag = 0) AND (one_sec_timer = 0))
            then one_sec_timer = 250;
            else if (one_sec_flag = 0)
```

```
                  then do;
                     decr one_sec_timer;
                     if (one_see_timer = 0) then one_sec_flag = 1;
                  end;
              if ((alarm_30 AND inhibit_bit <> 0) AND (alarm_30 AND panic<> 0)
                 then inhibit = 1;
                 else inhibit = 0;
              if (alarm_30 AND panic = 0)
                 then call knock_down;
           end;
        unsave regs;
   end adc_hndlr;


      _knock_down:
        stow_leds = led_out_latch = stow_leds AND $03;
        user_ld_req = inhibit = array_sel1 = array_sel2 = 0;
        strobe watchdog timer;
     end _knock_down;

;***************************************************************************
; This routine is the "main" routine that calls all of the routines that
; deal with the elements of calculating the state of charge of the battery.
; It is fully executed only when the dataset_ready_flag is set.  When all of
; these calculations are complete, this routine assigns a new value to
; bat_V_limit:
;***************************************************************************

Pseudocode:

battery_state_of_chg:
        save regs;
        call calc_sys_volts;
        call calc_state_of_chg;
        call calc_equal_count;
        if (state_of_chg < 1) OR (equal_count >0)
            then bat_V_limit = equal_V;
            else bat_V_limit = float_V;
        unsave regs;
    end battery_state_of_charge;

;***************************************************************************
; This routine takes the value of bat_temp, shifts it right by 4 subtracts
; $D and uses the result as an index into the each of three tables.  The
; first one is the float voltage table, the second one is the equalization
; voltage table and the last is the minimum voltage table.  This technique
; greatly reduces the execution time and simplifies the software.
;***************************************************************************

Pseudocode:

calc_sys_volts:
        save regs;
        index = bat_temp >> 4;
        index = index - $D;
        if (index < 0)
```

```
            then index = 0;
        index = index * 2;
        float_V = float_table[index];
        if (float_V > abs_max_bat_V)
            then float_V = abs_max_bat_V;
        equal_V = equal_table[index];
        if (equal_V > abs_max_bat_V)
            then equal_V = abs_max_bat_V;
        min_bat_V = min_bat_table[index];
        if (min_bat_V < abs_min_bat_V)
            then min_bat_V = abs_min_bat_V;
        unsave regs;
    end calc_sys_volts;
```

The following are the lookup tables for float_volts, equal_volts and min_bat_volts for the equation:

```
        value = base_voltage/cell * num_of_bat_cells *
                                    (1 + .0022 * (20.92 * (Vt - 2.56)))

                        where base_voltage/cell = 2.4 for float_V
                                                = 2.7 for equal_V
                                                = 1.9 for min_bat_V
```

In order to find the index to a value in the table, take the raw A/D value for Vt, shift right 4 bits and subtract $0D.

NOTICE: Vt, as shown below, is represented in "real volts" format, i.e. the number represents the number of 1/1000's of volts. To convert this value to what is actually stored in memory, it must be divided by 4.

FLOAT VOLTAGE TABLE

temp	index	Vt	hex value	dec value
61 deg C	00	340	012A	119.3
60 deg C	01	380	012B	119.7
58 deg C	02	3C0	012C	120.1
57 deg C	03	400	012D	120.4
56 deg C	04	440	012E	120.8
54 deg C	05	480	012F	121.2
53 deg C	06	4C0	012F	121.6
52 deg C	07	500	0130	122.0
50 deg C	08	540	0131	122.3
49 deg C	09	580	0132	122.7
48 deg C	0A	5C0	0133	123.1
46 deg C	0B	600	0134	123.5
45 deg C	0C	640	0135	123.9
44 deg C	0D	680	0136	124.3
42 deg C	0E	6C0	0137	124.6
41 deg C	0F	700	0138	125.0
40 deg C	10	740	0139	125.4
38 deg C	11	780	013A	125.8
37 deg C	12	7C0	013B	126.2
36 deg C	13	800	013C	126.5
34 deg C	14	840	013D	126.9

```
 33 deg C       15      880     013E    127.3
 32 deg C       16      8C0     013F    127.7
 30 deg C       17      900     0140    128.1
 29 deg C       18      940     0141    128.5
 28 deg C       19      980     0142    128.8
 26 deg C       1A      9C0     0143    129.2
 25 deg C       1B      A00     0144    129.6
 24 deg C       1C      A40     0144    130.0
 22 deg C       1D      A80     0145    130.4
 21 deg C       1E      AC0     0146    130.7
 20 deg C       1F      B00     0147    131.1
 18 deg C       20      B40     0148    131.5
 17 deg C       21      B80     0149    131.9
 16 deg C       22      BC0     014A    132.3
 14 deg C       23      C00     014B    132.7
 13 deg C       24      C40     014C    133.0
 12 deg C       25      C80     014D    133.4
 10 deg C       26      CC0     014E    133.8
  9 deg C       27      D00     014F    134.2
  8 deg C       28      D40     0150    134.6
  6 deg C       29      D80     0151    134.9
  5 deg C       2A      DC0     0152    135.3
  4 deg C       2B      E00     0153    135.7
  2 deg C       2C      E40     0154    136.1
  1 deg C       2D      E80     0155    136.5
 -0 deg C       2E      EC0     0156    136.9
 -2 deg C       2F      F00     0157    137.2
 -3 deg C       30      F40     0158    137.6
 -4 deg C       31      F80     0158    138.0
 -6 deg C       32      FC0     0159    138.4
 -7 deg C       33      1000    015A    138.8
 -8 deg C       34      1040    015B    139.1
-10 deg C       35      1080    015C    139.5

loat_table:
        .word   $012A, $012B, $012C, $012D, $012E, $012F, $012F, $0130
        .word   $0131, $0132, $0133, $0134, $0135, $0136, $0137, $0138
        .word   $0139, $013A, $013B, $013C, $013D, $013E, $013F, $0140
        .word   $0141, $0142, $0143, $0144, $0144, $0145, $0146, $0147
        .word   $0148, $0149, $014A, $014B, $014C, $014D, $014E, $014F
        .word   $0150, $0151, $0152, $0153, $0154, $0155, $0156, $0157
        .word   $0158, $0158, $0159, $015A, $015B, $015C

          EQUALIZATION VOLTAGE TABLE

  temp          index   Vt    hex value dec value

 61 deg C       00      340     0136    124.3
 60 deg C       01      380     0137    124.7
 58 deg C       02      3C0     0138    125.1
 57 deg C       03      400     0139    125.5
 56 deg C       04      440     013A    125.9
 54 deg C       05      480     013B    126.3
 53 deg C       06      4C0     013C    126.6
 52 deg C       07      500     013D    127.0
 50 deg C       08      540     013E    127.4
```

```
  49 deg  C       09       580      013F      127.8
  48 deg  C       0A       5C0      0140      128.2
  46 deg  C       0B       600      0141      128.6
  45 deg  C       0C       640      0142      129.0
  44 deg  C       0D       680      0143      129.4
  42 deg  C       0E       6C0      0144      129.8
  41 deg  C       0F       700      0145      130.2
  40 deg  C       10       740      0146      130.6
  38 deg  C       11       780      0147      131.0
  37 deg  C       12       7C0      0148      131.4
  36 deg  C       13       800      0149      131.8
  34 deg  C       14       840      014A      132.2
  33 deg  C       15       880      014B      132.6
  32 deg  C       16       8C0      014C      133.0
  30 deg  C       17       900      014D      133.4
  29 deg  C       18       940      014E      133.8
  28 deg  C       19       980      014F      134.2
  26 deg  C       1A       9C0      0150      134.6
  25 deg  C       1B       A00      0151      135.0
  24 deg  C       1C       A40      0152      135.4
  22 deg  C       1D       A80      0153      135.8
  21 deg  C       1E       AC0      0154      136.2
  20 deg  C       1F       B00      0155      136.6
  18 deg  C       20       B40      0156      137.0
  17 deg  C       21       B80      0157      137.4
  16 deg  C       22       BC0      0158      137.8
  14 deg  C       23       C00      0159      138.2
  13 deg  C       24       C40      015A      138.6
  12 deg  C       25       C80      015B      139.0
  10 deg  C       26       CC0      015C      139.4
   9 deg  C       27       D00      015D      139.8
   8 deg  C       28       D40      015E      140.2
   6 deg  C       29       D80      015F      140.6
   5 deg  C       2A       DC0      0160      141.0
   4 deg  C       2B       E00      0161      141.4
   2 deg  C       2C       E40      0162      141.8
   1 deg  C       2D       E80      0163      142.2
  -0 deg  C       2E       EC0      0164      142.6
  -2 deg  C       2F       F00      0165      143.0
  -3 deg  C       30       F40      0166      143.4
  -4 deg  C       31       F80      0167      143.7
  -6 deg  C       32       FC0      0168      144.1
  -7 deg  C       33       1000     0169      144.5
  -8 deg  C       34       1040     016A      144.9
 -10 deg  C       35       1080     016B      145.3

equal_table:

        .word   $0136, $0137, $0138, $0139, $013A, $013B, $013C, $013D
        .word   $013E, $013F, $0140, $0141, $0142, $0143, $0144, $0145
        .word   $0146, $0147, $0148, $0149, $014A, $014B, $014C, $014D
        .word   $014E, $014F, $0150, $0151, $0152, $0153, $0154, $0155
        .word   $0156, $0157, $0158, $0159, $015A, $015B, $015C, $015D
        .word   $015E, $015F, $0160, $0161, $0162, $0163, $0164, $0165
        .word   $0166, $0167, $0168, $0169, $016A, $016B
```

MINIMUM BATTERY VOLTAGE TABLE

temp	index	Vt	hex value	dec value
61 deg C	00	340	00EC	94.4
60 deg C	01	380	00EC	94.7
58 deg C	02	3C0	00ED	95.0
57 deg C	03	400	00EE	95.3
56 deg C	04	440	00EF	95.6
54 deg C	05	480	00EF	96.0
53 deg C	06	4C0	00F0	96.3
52 deg C	07	500	00F1	96.6
50 deg C	08	540	00F2	96.9
49 deg C	09	580	00F2	97.2
48 deg C	0A	5C0	00F3	97.5
46 deg C	0B	600	00F4	97.8
45 deg C	0C	640	00F5	98.1
44 deg C	0D	680	00F5	98.4
42 deg C	0E	6C0	00F6	98.7
41 deg C	0F	700	00F7	99.0
40 deg C	10	740	00F8	99.3
38 deg C	11	780	00F8	99.6
37 deg C	12	7C0	00F9	99.9
36 deg C	13	800	00FA	100.2
34 deg C	14	840	00FB	100.5
33 deg C	15	880	00FB	100.8
32 deg C	16	8C0	00FC	101.1
30 deg C	17	900	00FD	101.4
29 deg C	18	940	00FE	101.7
28 deg C	19	980	00FE	102.0
26 deg C	1A	9C0	00FF	102.3
25 deg C	1B	A00	0100	102.6
24 deg C	1C	A40	0101	102.9
22 deg C	1D	A80	0102	103.2
21 deg C	1E	AC0	0102	103.5
20 deg C	1F	B00	0103	103.8
18 deg C	20	B40	0104	104.1
17 deg C	21	B80	0105	104.4
16 deg C	22	BC0	0105	104.7
14 deg C	23	C00	0106	105.0
13 deg C	24	C40	0107	105.3
12 deg C	25	C80	0108	105.6
10 deg C	26	CC0	0108	105.9
9 deg C	27	D00	0109	106.2
8 deg C	28	D40	010A	106.5
6 deg C	29	D80	010B	106.8
5 deg C	2A	DC0	010B	107.1
4 deg C	2B	E00	010C	107.4
2 deg C	2C	E40	010D	107.7
1 deg C	2D	E80	010E	108.0
-0 deg C	2E	EC0	010E	108.3
-2 deg C	2F	F00	010F	108.6
-3 deg C	30	F40	0110	108.9
-4 deg C	31	F80	0111	109.2
-6 deg C	32	FC0	0111	109.6
-7 deg C	33	1000	0112	109.9

```
  -8 deg C       34        1040      0113      110.2
 -10 deg C       35        1080      0114      110.5

min_bat_table:

        .word   $00EC, $00EC, $00ED, $00EE, $00EF, $00EF, $00F0, $00F1
        .word   $00F2, $00F2, $00F3, $00F4, $00F5, $00F5, $00F6, $00F7
        .word   $00F8, $00F8, $00F9, $00FA, $00FB, $00FB, $00FC, $00FD
        .word   $00FE, $00FE, $00FF, $0100, $0101, $0102, $0102, $0103
        .word   $0104, $0105, $0105, $0106, $0107, $0108, $0108, $0109
        .word   $010A, $010B, $010B, $010C, $010D, $010E, $010E, $010F
        .word   $0110, $0111, $0111, $0112, $0113, $0114

;*******************************************************************
; This routine is called to calculate the running "sum of coulombs" whic
; the battery state of charge.  The basic algorithm calls for the follow
; calculation to be performed:
;
;       state_of_chg = state_of_chg + (battery_I * iscal /battery_cap)
;
; The basic problem here is dealing with the extremely large range of th
; numbers involved. State_of_chg is stored as a 40 bit number with a
; maximum value of 1. (followed by 39 zeros).  For values of iscal and
; battery capacity of 1/45000 and 400, respectively, and battery_I store
; the form, xxx.x, this equation reduces to:
;
;       state_of_chg = state_of_charge + (555 * battery_I * (10)^-11)
;
; If, for the purposes of calculation, we make the binary point right ju
; by multiplying thru by (2)^39, the equation becomes:
;
; (5.49 * (10)^11) * state_of_chg =
;                               (5.49 * (10)^11) * state_of_chg +
;                                   (5.49 * (10)^11) * (555 * battery_I * (10)^-1
;
; Which reduces further, in general, to:
;
; (5.49 * (10)^11) * state_of_chg =
;                                       (5.49 * (10)^11) * state_of_chg +
;                                               (iscal_bat_cap * battery_I)
;
; Hence, to calculate a new value of state_of_chg, multiply battery_I by
; parameterized value, iscal_bat_cap, which is equal to:
;                 5.49 * 10^10 * iscal * 1/bat_cap.
; and add (or subtract) the result to (or from) the running 40 bit value
; state_of_chg.
;
; To read the value of soc, mentally move the binary point left to the or
; position (i.e. divide by (2)^39) and read the value.
;
; If the battery is charging, i.e. the net current is flowing INTO the ba
; the battery current is multiplied by the coulombic efficiency which is
; function of the state_of_chg as follows:
;
;                 State_of_chg                              coulombic_eff
;       decimal             binary                     decimal          binary
```

```
;        0 to .6874        < 0 1011000              1.00000              -
;    .6875 to .8124        < 0 1100000               .9063            .11101
;    .8125 to .9062        < 0 1110100               .8125            .11010
;    .9063 to 1.000        < 1 0000000               .6875            .10110
;****************************************************************************

Pseudocode:

calc_state_of_chg:
        save regs;
        if (battery_I >0)
          then do;
            if (state_of_chg >= soc_brk1)
              then coulombic_eff = coul_eff1;
              else if ((state_of_chg >= soc_brk2)
                                  AND (state_of_charge < soc_brk1))
                then coulombic_eff = coul_eff2;
                else if ((state_of_chg >= soc_brk3)
                                  AND (state_of_charge < soc_brk2))
                  then coulombic_eff = coul_eff3;
                  else coulombic_eff = 1.0;
            battery_I = battery_I * coulombic_eff;
          end;
        state_of_chg = state_of_chg + (battery_I * iscal_bat_cap);  see above
        if (state_of_chg > 1)
            then state_of_chg = 1;
        if (state_of_chg < 0)
            then state_of_chg = 0;
        unsave regs;
    end calc_state_of_chg;

;****************************************************************************
; This routine is called to calculate a running total of equalization counts.
; For details about dealing with the large (40 bits!) equalization count,
; see calc_state_of_chg.
; Since there was och_frac to deal with, a section of the code for this
; routine is a "hardwired" multiply routine so that it wouldn't be necessary
; to use the long 16 by 16 multiply routine.  It is assumed, in order to make
; this routine as fast as possible, that the fraction is only 7 bits long,
; and that it is left justified with the binary point at the left end, like
; a fraction should be.  In order to minimize the code and the requirement
; for additional zero page storage locations for the interim solutions, some
; of the variable locations used in mp_mult are also used here, as shown belc
;
;                                                        temp_och_frac
;                                                       /
;       | iscal_bat_cap |        bat_mult2        |   /     |
;       |_______|_______|_______|_______|_______|_______|_______|_______|
;       |       |       |       |       |       |       |       |       |
;       |     mult1     |     mult2     |            product            |
;
;  The product of bat_mult2 * och_frac is shifted as it is multiplied, into
;  the two bytes occupied by mult2. Temp_och_frac is overwritten.
;****************************************************************************

Pseudocode:
```

```
calc_equal_count:
        save regs;
        if (battery_I < 0)
            then do;
                temp = (battery_I * och_frac)/256;
                equal_count = equal_count + (temp * iscal_bat_cap);
            end;
        if (state_of_chg >= 1)
                equal_count = equal_count - (battery_I * iscal_bat_cap)
            then state_of_chg = 1;
        if (equal_count < 0)
            then equal_count = 0;
        if (equal_count > max_equal_count)
            then equal_count = max_equal_count;
        unsave regs;
    end calc_equal_count;

;*********************************************************************
; This routine corrects the current state_of_chg value for battery
; temperature.
;*********************************************************************
```

Pseudocode:

```
correct_state_of_chg:
        save regs;
        if ((abs_battery_V >= .99 * equal_V) OR (state_of_chg > soc_top
            then do;
                state_of_chg = state_of_chg + .01;
                if (state_of_chg > 1.0) then state_of_chg = 1.0;
            end;
        if (abs_battery_V < min_bat_V)
            then do;
                state_of_chg = state_of_chg - .01;
                if (state_of_chg < 0) then state_of_chg = 0;
            end;
        index = (bat_temp >> 4) - $D;
        if (index < 0)
            then index = 0;
        cstate_of_chg = state_of_chg * csoc_table[index];
        unsave regs;
    end correct_state_of_chg;
```

The following are the lookup table for the values in the expression:

```
        value = (1 + coeff * (20.92 * (Vt - 2.56)))

                        where coeff = .0022 for temperatures > 25 deg (
                                    = .0075 for temperatures < 25 deg (
```

In order to find the index to a value in the table, take the raw A/D value for Vt, shift right 4 bits and subtract $0D.

NOTICE 1: The value stored in the table is assumed to have a hexadecima point in the center of the word, e.g. $0112 corresponds to $1.12

NOTICE 2: Vt, as shown below, is represented in "real volts" format, i.e. the number represents the number of 1/1000's of volts. To convert this value to what is actually stored in memory, it must be divided by 4.

CORRECT STATE OF CHARGE TABLE

temp	index	Vt	hex value	percent
61 deg C	00	340	0114	108%
60 deg C	01	380	0113	108%
58 deg C	02	3C0	0112	107%
57 deg C	03	400	0112	107%
56 deg C	04	440	0111	107%
54 deg C	05	480	0110	106%
53 deg C	06	4C0	010F	106%
52 deg C	07	500	010F	106%
50 deg C	08	540	010E	106%
49 deg C	09	580	010D	105%
48 deg C	0A	5C0	010C	105%
46 deg C	0B	600	010C	105%
45 deg C	0C	640	010B	104%
44 deg C	0D	680	010A	104%
42 deg C	0E	6C0	0109	104%
41 deg C	0F	700	0109	104%
40 deg C	10	740	0108	103%
38 deg C	11	780	0107	103%
37 deg C	12	7C0	0106	103%
36 deg C	13	800	0106	102%
34 deg C	14	840	0105	102%
33 deg C	15	880	0104	102%
32 deg C	16	8C0	0103	101%
30 deg C	17	900	0103	101%
29 deg C	18	940	0102	101%
28 deg C	19	980	0101	101%
26 deg C	1A	9C0	0100	100%
25 deg C	1B	A00	0100	100%
24 deg C	1C	A40	00FD	99%
22 deg C	1D	A80	00FA	98%
21 deg C	1E	AC0	00F8	97%
20 deg C	1F	B00	00F5	96%
18 deg C	20	B40	00F3	95%
17 deg C	21	B80	00F0	94%
16 deg C	22	BC0	00EE	93%
14 deg C	23	C00	00EB	92%
13 deg C	24	C40	00E8	91%
12 deg C	25	C80	00E6	90%
10 deg C	26	CC0	00E3	89%
9 deg C	27	D00	00E1	88%
8 deg C	28	D40	00DE	87%
6 deg C	29	D80	00DC	86%
5 deg C	2A	DC0	00D9	85%
4 deg C	2B	E00	00D6	84%
2 deg C	2C	E40	00D4	83%
1 deg C	2D	E80	00D1	82%

```
  -0 deg C        2E        EC0       00CF        81%
  -2 deg C        2F        F00       00CC        80%
  -3 deg C        30        F40       00CA        79%
  -4 deg C        31        F80       00C7        78%
  -6 deg C        32        FC0       00C4        77%
  -7 deg C        33        1000      00C2        76%
  -8 deg C        34        1040      00BF        75%
 -10 deg C        35        1080      00BD        74%

csoc_table:

        .word   $0114, $0113, $0112, $0112, $0111, $0110, $010F, $010F
        .word   $010E, $010D, $010C, $010C, $010B, $010A, $0109, $0109
        .word   $0108, $0107, $0106, $0106, $0105, $0104, $0103, $0103
        .word   $0102, $0101, $0100, $0100, $00FD, $00FA, $00F8, $00F5
        .word   $00F3, $00F0, $00EE, $00EB, $00E8, $00E6, $00E3, $00E1
        .word   $00DE, $00DC, $00D9, $00D6, $00D4, $00D1, $00CF, $00CC
        .word   $00CA, $00C7, $00C4, $00C2, $00BF, $00BD

;**********************************************************************
; This routine, called once per second, performs the actual control of t
; loads connected to the system, as determined by the state variables
; prod_mask (created by determine_mach_state), user_ld_req (set by the u
; from the terminal or keypad), and overld_trip (set by chk_for_overld).
; Depending upon the state of charge of the battery (as indicated in
; cstate_of_chg), various loads are turned on or off so as to maximize
; battery life.  In addition, various warning indicators (RED and YELLOW
; LEDs, and the Low Battery indication on the LCD Display) are turned on
; or off to alert the operator to unusual or dangerous conditions.
;**********************************************************************

Pseudocode:

shed_restor_loads:
        save regs;
        if ((hours >= #$08) AND (hours < #$12) AND
                (total_chgr_I > (5.12 * 0.1 * *num_load_strings)))
            then delta = #delta_soc;    /* delta_soc = 0.1 */
            else delta = 0;
        j = stow_leds;
        do i = 4 to 0 step -1;
            if ((cstate_of_chg + delta) < shed_thresh[i])
                then do;
                    j = j AND shed_msk_tbl[i];
                    ignore the next statement and continue loop;
                end;
            if ((cstate_of_chg + delta) >= restor_thresh[i])
                    j = j OR restor_msk_tbl[i];
        end;
        j = (j OR #ld6_on) AND prod_mask);
        if ((j & restor_msk_tbl[4]) <> 0)
            then j = j AND #yled_off;
            else j = j OR #yled_on;
        if ((j AND restor_msk_tbl[0]) <> 0)
            then j = j AND #rled_off;
            else do;
```

```
                j = j OR #rled_on;
                j = j AND #yled_off;
            end;
        sr_temp = j;
        j = user_ld_req OR #leds_on;
        sr_temp = sr_temp AND j;
        if ((sr_temp & #rled_on) <> 0)
            then alarm_flags = alarm_flags OR #enable_lobat;
            else alarm_flags = alarm_flags AND #disable_lobat;
        if (ovrld_trip <> 0)
            then do;
                j = sr_temp AND ~ovrld_trip;
                stow_leds = j | #rled_on;
            end;
            else stow_leds = sr_temp;
        led_out_latch = stow_leds;
        unsave regs;
    end shed_restor_loads;

;**************************************************************************
; This routine, called every second, checks the currents in the 5 load
; strings and if a current exceeds a limit for more than a certain number
; of times (this "count" being load specific), the bit corresponding to
; the load number that has suffered the overload is set in the ovrld_trip
: mask, and the load is immediately turned off (as opposed to waiting for
: the next execution of the shed/restore load routine).
;**************************************************************************

'seudocode:

 hk_for_ovrld:
        save regs;
        do i = 4 to 0 step -1;
            if (bus_amps[i] < 0)
                then sr_temp = - bus_amps[i];
                else sr_temp = bus_amps[i];
            if (sr_temp > ovrld_thresh[i])
                then do;
                    ovrld_cnt[i] = ovrld_cnt[i] - 1;
                    if (ovrld_cnt[i] = 0)
                        then ovrld_trip = ovrld_trip OR ovrld_trip_msk[i];
                end;
                else ovrld_cnt[i] = ovrld_cnt_max[i];
        end;
        if (ovrld_trip <> 0)
            then do;
                stow_leds = stow_leds AND ~ovrld_trip;
                stow_leds = stow_leds OR #rled_on;
                led_out_latch = stow_leds;
                alarm_flags = alarm_flags OR #enable_bell;
            end;
        unsave regs;
    end chk_for_ovrld;

 **************************************************************************
 The function of this routine is to determine if more or less array branches
```

```
; should be connected.  This routine is entered every time the one_sec_f
; is set.  The pattern for the two PWM latches are held in bits 0 thru 6
; stow_pwm1 and stow_pwm2.
;***********************************************************************

Pseudocode:

discrete_array_cntrl:
        save reg;
        if (abs_battery_V > bat_V_limit)
            then do;
                stow_pwm1 = 0;
                stow_pwm2 = 0;
            end;
            else if (abs_battery_V > (bat_V_limit * 31)/32)
                then do;
                    if (stow_pwm2 = 0)
                        then do;
                            stow_pwm1 = stow_pwm1 >> 1;
                            stow_pwm2 = stow_pwm2 >> 1;
                        end;
                end;
                else if (abs_battery_V < (bat_V_limit * 30)/32)
                    then do;
                        stow_pwm1 = (stow_pwm1 << 1) + 1;
                        if (stow_pwm1 AND $40 <> 0)
                            then do;
                                stow_pwm1 = stow_pwm1 AND $3F;
                                stow_pwm2 = (stow_pwm2 << 1) + 1;
                                stow_pwm2 = stow_pwm2 AND $3F;
                            end;
                    end;
        array_sel1 = stow_pwm1;
        array_sel2 = stow_pwm2;
        unsave regs;
    end discrete_array_cntrl;

;***********************************************************************
; The routine determines which of the four states of the machine (descri
; below) that the machine should be in based on state_of_chg and product
; levels and then sets up those conditions.  It is assumed that the prio
; of functions is:
;
;                1. charging the battery
;                2. running the load
;                3. making product
;
; state #1       battery is so low that product and load relays are all
;                off (state1  = 00000000)
;
; state #2       battery partially charged, load relays closed,
;                product relay open  (state2 = 11111000)
;
; state #3       battery close to full charge, load relay open,
;                product relay closed (state3 = 00000100)
;
```

```
; state #4       battery close to full charge, load relays closed,
;                product relay closed (state4 = 11111100)
;
;       note 1: When refering to the fact that the load relays are closed, t
;               presumes that the overload threshold has not been exceeded.
;
;       note 2: The product relay is load #1, the load relays operate as a
;               unit (in so far as this routine is concerned), so that
;               loads 2 thru 5 are turned off and on together.
;
; State Diagram:
;                                (00)
;                          ---->STATE 1-->--
;                         |                 |
;                         ^                 |
;                         |                 v
;                  (corr_soc<soc1)          |
;                         |       (corr_soc>soc1+bufsoc1)
;                         |                 |
;                         ^                 v
;                          \               /
;                           \             /
;                            \           /
;                             \  (F8)   /
;          -<----<----<---<--<--STATE 2<---->--->---->---->-----
;         |                     /    \                          |
;         v                    /      \                         v
;    (prod<prod1)             /        \                (prod1<prod<prod2)
;        AND                 |          \                      AND
;(corr_soc>soc2+bufsoc2)     ^           |           (corr_soc>soc3+bufsoc3)
;         |                  |           ^                      |
;         |       (corr_soc<soc2)        |                      v
;         |                  ^      (prod>prod2+bufprod2)       |
;         v                  |                 OR               |
;         |     --->---->----           (corr_soc<soc3)         |
;         |    /                                ^               v
;       STATE 3<----<----<--                    |               |
;         | (04)            |                    -<----<----<-  |
;         |            (prod<prod1)                           | |
;         |                AND      <----<----<----<----<----STATE 4
;         |          (corr_soc<soc4)                            | (FC)
;         v                                                     ^
;         |                                                     |
;          ---->(prod>prod1+bufprod1)OR(corr_soc>soc4+bufsoc4)--
;
;*************************************************************************

Pseudocode:

deter_mach_state:
   save regs;
   if (prod_mask = state1)
       then if (cstate_of_chg > soc1 + bufsoc1)          ***case1***
          then prod_mask = state2;
       else if (prod_mask = state2)
          then do;
```

```
            if (cstate_of_chg < soc1)                        ***case2***
               then prod_mask = state1;
               else if ((prod1 <= prod < prod2)
                    AND (cstate_of_chg > soc3 + bufsoc3)) ***case3*
                  then prod_mask = state4;
                  else if ((prod < prod1)
                         AND (cstate_of_chg > soc2 + bufsoc2)) **ca
                     then prod_mask = state3;
         end;
         else if (prod_mask = state3)
            then do;
               if (cstate_of_chg < soc2)                    ***case5***
                  then prod_mask = state2;
                  else if ((prod > prod1 + bufprod1) OR
                          (cstate_of_chg > soc4 + bufsoc4)  ***case6*
                     then prod_mask = state4
            end;
            else if (prod_mask = state4)
               then do;
                  if ((prod < prod1) AND (cstate_of_chg < soc4))
                     then prod_mask = state3;             ***case7***
                     else if ((prod > prod2 + bufprod2)
                              OR (cstate_of_chg < soc3))  ***case8***
                        then prod_mask = state2;
               end;
      call shed_restor_loads;
      unsave regs;
   end deter_mach_state;

;***********************************************************************
; This routine handles all of the initialization duties, runs the RAM
; and lamp test.
;***********************************************************************

Pseudocode:

test:

      disable interrupts;
      make sure we're in decimal mode;
      stack pointer = $FF;
      pia_csra = ddr_sel;
      pia_csrb = ddr_sel;
      pia_ddra = paddr_mask;
      pia_csra = pa_edge_sel OR ddr_desel;
      pia_porta = dsply_sel_dsb AND dsply_clk_off;
      pia_ddrb = pbddr_mask;
      pia_csrb = pb_edge_sel OR ddr_desel;
      pia_portb = $0F;
      led_out_latch = $00;
      do x = 0 to $FF
          $0000[x] = $FF;
      end;
      do x = 0 to $FF
          if ($0000[x] <> $FF)
              then call ram_error;
```

```
        end;
        do x = 0 to $FF
                $0000[x] = $00;
        end;
        do x = 0 to $FF
            if ($0000[x] <> $00)
                then call ram_error;
        end;
        rom_ptr = $0100;
        do while (rom_ptr[1] < max_ram_size)
            $0000[rom_ptr] = $FF;
            rom_ptr = rom_ptr + 1;
        end;
        rom_ptr = $0100;
        do while (rom_ptr[1] < max_ram_size)
            if ($0000[rom_ptr] <> $FF)
                then call ram_error;
            rom_ptr = rom_ptr + 1;
        end;
        rom_ptr = $0100;
        do while (rom_ptr[1] < max_ram_size)
            $0000[rom_ptr] = $00;
            rom_ptr = rom_ptr + 1;
        end;
        rom_ptr = #$0100;
        do while (rom_ptr[1] < max_ram_size)
            if ($0000[rom_ptr] <> $00)
                then call ram_error;
            rom_ptr = rom_ptr + 1;
        end;
        pia_portb = bell_on OR $0F;
        call lamp_test;
        pia_portb = $0F;
nit_timer:
        timer_csr = timer_reset;
        timer_csr = ldall_timers;
        timer_csr = $0F;
        timer_data = $00;
        timer_data = $00;
        timer_data = master_model;
        timer_data = master_modeh;
        timer_csr = $01;
imer1:
        timer_data = cntr1_model;
        timer_data = cntr1_modeh;
        timer_data = $00;
        timer_data = $00;
        timer_data = $00;
        timer_data = $00;
imer2:
        timer_data = cntr2_model;
        timer_data = cntr2_modeh;
        timer_data = $00;
        timer_data = $00;
        timer_data = $00;
        timer_data = $00;
```

```
        timer_csr = $43;
        timer_csr = $0A;
        timer_data = $59;
        timer_data = $23;
        timer_data = $00;
        timer_data = $00;
timer3:
        timer_data = cntr3_model;
        timer_data = cntr3_modeh;
        timer_data = cntr3_ld_reg;
        timer_data = cntr3_ld_reg+1;
        timer_data = cntr3_hold_reg;
        timer_data = cntr3_hold_reg+1;
timer4:
        timer_data = cntr4_model;
        timer_data = cntr4_modeh;
        timer_data = cntr4_ld_reg;
        timer_data = cntr4_ld_reg+1;
        timer_data = cntr4_hold_reg;
        timer_data = cntr4_hold_reg+1;
timer5:
        timer_data = cntr5_model;
        timer_data = cntr5_modeh;
        timer_data = def_baud_rate;
        timer_data = def_baud_rate+1;
start_timers:
        timer_csr = $73;
        timer_csr = $0A;
        timer_data = $00;
        timer_data = $00;
        timer_csr = $E3;
        timer_csr = $E4;
        timer_csr = $2C;
init_zpg:
        compute_flags = compute_flags OR run_flag;
        sp_stor = $FF;
        column_number = $08;
        pia_portb = column_number AND $0F;
        write_portb = column_number AND $0F;
        bounce_count = max_bounce_count;
        stow_leds = $00;
        led_out_latch = $00;
        stow_pwm1 = $00;
        array_sel1 = $00;
        stow_pwm2 = $00;
        array_sel2 = $00;
        next_seldom = $18;
        do x = 4 to 0 step -1
            overld_cnt[x] = overld_cnt_max[x];
            shed_thresh[x] = shed_th[x];
            restor_thresh[x] = restor_th[x];
        end;
        state_of_chg[4] = $40;
        inhibit = $01;
        one_sec_timer = 250;
        call msg_hndlr('WELCOME TO THE TRISOLAR DEBUG MONITOR');
```

```
        call display_clr;
        i_o_flags = i_o_flags OR dataset_ready_flag;
        pia_csra = pa_edge_sel OR ddr_desel OR x4_msec_enab;
        enable interrupts;
        goto run_task_master;

ram_error:

        call display_error(2);
        call display_digits;
    loop:
        goto loop;
    end ram_error;

;**************************************************************************
: This routine is called every 100 msec to average the 16 readings of each
: of the 12 often_read channels.
;**************************************************************************

'suedocode:

signal_av:
       do next_often = 0 to num_of_often_reads;
          adc_temp_hold = 0;
          do pointer = 0 to max_pass_num;
             adc_temp_hold = often_read[pointer,next_often*32]
                                        + adc_temp_hold;
          end;
           often_read[next_often] = adc_temp_hold/16;
        end;
        total_chgr_I = 0;
           do index = 2 to num_of_often_reads;
              total_chgr_I = dump_array[index] + total_chgr_I;
           end;
    end signal_av;
```

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-174723		

4. Title and Subtitle
Development of a Microprocessor Controller for Stand-Along Photovoltaic Power Systems

5. Report Date
June, 1984

6. Performing Organization Code
TSC 210F

7. Author(s)
Alan R. Millner

8. Performing Organization Report No.

9. Performing Organization Name and Address
TriSolarCorp
10 De Angelo Drive
Bedford, MA 01730

10. Work Unit No.

11. Contract or Grant No.
DEN3-310

12. Sponsoring Agency Name and Address
U.S. Department of Energy
Division of Photovoltaic Energy Technology
Washington, D.C. 20545

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Report Number
DOE/NASA/0310-1

15. Supplementary Notes
Final Report. Prepared under Interagency Agreement DE-AI01-79ET20485. Project Manager, R. DeLonbard, Energy Technology Division, NASA Lewis Research Center, Cleveland, Ohio 44135.

16. Abstract
A controller for stand-alone photovoltaic systems has been developed using a low power CMOS microprocessor. It performs battery state of charge estimation, array control, load management, instrumentation, automatic testing, and communications functions. Array control options are sequential subarray switching and maximum power control. A calculator keypad and LCD display provides manual control, fault diagnosis and digital multimeter functions. An RS-232 port provides data logging or remote control capability. A prototype 5kW unit has been built and tested successfully. The controller is expected to be useful in village photovoltaic power systems, large solar water pumping installations, and other battery management applications.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Photovoltaic, Controller, Microprocessor, Battery, State-of-Charge, Load Management	Unclassified-unlimited STAR Category 44 DOE Category UC-63

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	160	

*For sale by the National Technical Information Service, Springfield, Virginia 22161