A REAL_TIME INK CORRECTION MODULE FOR HELIO_ENGRAVING PROCESS

by

SUDHINDRA NATH MISHRA

B. Sc. Engg., Ranchi University, India
(1968)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE

at the

February, 1981

Signature of Author..
Department of Electrical Engg. and Computer Sc., January 22, 1981

Certified by..

Accepted by..
Chairman, Department Committee

A REAL_TIME INK CORRECTION MODULE FOR HELIO_ENGRAVING PROCESS

by

SUDHINDRA NATH MISHRA

submitted to the department of Electrical Engineering and Computer Science on January 22, 1981, in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

A Computer_aided Color Image Processing system for a rotogravure engraving machine, called Helio_Klischograph, is currently under development. The color processing would consist of two phases viz. Color Translation Process and Helio-Engraving Process. Color translation is more conveniently done in terms of ICI values, using a TV display, rather than ink or dye_densities. Significant storage efficiency is obtained if the translated image is represented in luminance_chrominance space. The transformation between ICI values and luminance_chrominance representation is conveniently provided by simple linear transformation using a 3X3 matrix. However, the transformation between the ICI values and the ink_densities is non_linear, necessitating a piecewise linear approximation approach. A procedure using sub_resolution correction_lookup_table is proposed for the computation of ink_densities from the ICI values. The correction_table is determined empirically by matching ink_patches to the ICI values.

Storage economy demands that the computation of ink_densities from ICI values must be done in real_time, while Helio_Klischograph is engraving rotogravure cylinders. For this real_time computation, a high_speed digital processor is required. A specially architectured microprogrammable machine was developed to fulfill this need. The real_time processor is managed by a resident microcomputer through software, yielding a highly flexible and computationally powerful system, called Ink Correction Module. ICM also features high_speed interprocessor communication with PDP-11 minicomputer family. A well structured top_down design approach was used for both hardware and software.

Thesis Supervisor: William F. Schreiber
Title: Professor of Electrical Engineering

Acknowledgement

I sincerely wish to express my gratitude to all those who helped me during the course of this project.

My deepest thanks go to Professor William F. Schreiber, for his generous support, patient encouragement and constant guidance. He has opened a new field of knowledge for me.

I also thank Professor Donald E. Troxel for his support and guidance.

Thanks are also due to my student colleague Gary Neben, who helped me with the construction of the HHKD module. Special thanks to Rich Daemon with whom I had many useful discussions.

I also wish to express the pleasure of working in an ideal environment provided by the CIPG Laboratory.

Most of all, I appreciate the participation of my wife Neelu, who sat in front of my computer terminal for long hours and handled the entire text_proceesing of this dissertation. Her love, understanding and companionship were even more valuable.

Last but not the least, I must acknowledge the contribution of my two lovely children Rajesh and Shivani, who provided me the entertainment I needed most after long hours of work.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1

Introduction

Two methods are predominantly used for printing images in the graphic arts industry. "Intaglio" is the generic name given to processes in which scratches (depressions) in surfaces, most often metal, are filled with ink which is then transferred to paper by pressing the latter against the surface with a soft pad. Usually the entire object is first inked and the surface wiped clean. Wood block printing, the other method, is in a sense opposite of engraving, in which, material is removed from the block where no ink is to be printed instead of vice versa. The relief surface is inked and the image transferred in much the same manner.

Both, relief printing (now called letterpress) and Intaglio (now "gravure" or "Rotogravure"), have undergone centuries of improvements and refinements to arrive at the current state of technology. But the Intaglio plates have been inherently capable of better detail, since an inked line corresponds to a depression (scratch) rather than a ridge. In particular, the ability to make very thin lines made it possible to simulate shadings of tone by regulating line spacing.

A significant step in both types of plate making was the discovery that material could be removed by acid etching. With the advent of photographic techniques in late 19th century, the acid etching method (now called "photolithography") could produce remarkable results chiefly due to accurate control on "masking" provided by use of photo_resist and optical imaging. However, "engraving" (implying material removal by mechanical means such as by a cutting tool, as opposed to material removal by chemical means as implied by the term "etching") survived as an extremely potent and viable method for Intaglio plate making chiefly due to its simplicity, cleanliness, cost of production and scope for future development. While the traditional forms of Intaglio are still used for art prints, the modern mechanised form has become of great commercial importance. This process, often called photogravure or rotogravure, is, by some accounts, the fastest growing form of printing [12].

1.1. Printing by Rotogravure

Rotogravure printing is carried out on high_speed, web_fed rotary presses [12]. The printing surface is in the form of a cylinder, generally copper and often very large, having an array of small etched or engraved cells, typically 150 to 200 per inch. The cylinder is rotated in a bath of ink and is wiped clean by a "doctor blade" as the surface emerges. Paper is then fed against the cylindrical surface, picking up the ink from the cells. The ink density of every cell on paper depends upon the quantity of ink transferred from the corresponding cell on gravure cylinder and thus depends upon its geometry. By modulating the cell_size on gravure surface, it is then possible to modulate the ink_density of corresponding dot on paper, thus forming a half_tone image. The big advantage of this process is that it can be used to make millions of impressions with very accurate metering of ink. Since the paper does not come in contact with the inside walls of the cell, they do not wear out. The surface of the cylinder, which does wear down, may be repeatedly replated with chromium. Also, the accurate ink transfer which is virtually independent of speed, makes this process very suitable for color printing.

The color pages are printed by overlaying multiple images, one in each primary color ink, so that their combination achieves the desired result. The primary color inks used in the printing industry are Yellow, Cyan (also called Precess_Blue) and Magenta (also called Process_Red). Although, in principle, it should be possible to generate any arbitrary color (within the limitations of ink's gamut ofcourse) by combining only these three primary colors, in practice, however, a Black printer (also called the "Key") is often included for reasons discussed in section 2.4. The etching or engraving of cells in the gravure cylinder implies that the printed image is "screened". This reduces the sharpness of the print. The presence of screen is not very objectionable due to a number of reasons and with a proper choice, highly acceptable results are obtained.

1.2. The Helio Klischograph

The Helio_Klischograph [1], made by Hell, engraves the cells in gravure cylinders by means of a battery of diamond styli which operate at 3600 to 4000 cells per second. For a typical cylinder eight feet long and forty_three inches in circumference and capable of printing thirty_two magazine pages, eight styli are spaced along the cylinder. Each moves in and out, cutting four pages as the cylinder rotates, engraving the entire cylinder in about an hour. Because of the geometry of the Helio, only diamond_shaped cells can be cut. As a result, it is not possible for the cell shape to conform to type stroke boundaries as well as it does in etched cylinder. However, this is compensated for by the cleanliness, speed and consistency of the process. Moreover, it has been found that by coding "type" and "lineart" areas differently than tone [2], highly acceptable results are produced.

Specially prepared images called "Cronapaques" are mounted on a scanning drum which rotates in synchronism with the cylinder to be engraved. Optical sensors mounted on the scanning drum provide the video information to the diamond stylus engraving heads. In order to distinguish between "dropout" which must occur for the white portion of the type or lineart, the minimum density of the continuous_tone material must be accurately controlled. This necessitates a large number of photographic steps (between five and ten) that must be accomplished in order to produce Cronapaques to be mounted on the scanning drum. These photographic steps contribute singnificantly to both the time and cost of the production of gravure cylinders. Inevitably, some loss of quality is incurred because of the large number of photographic steps involved in the current technology. These considerations motivated the development of a computer_based automated engraving system.

1.3. Automated Engraving of Gravure Cylinders

A computer_based system for automated engraving of gravure cylinders [2] has been developed and is now being used in normal production environment. Either fully composed pages or individual page components are scanned and stored on a large disk. In the case of fully composed pages, an operator uses a TV_display to segment the page_image into line and tone areas. The image is then coded by a software process and is ready for subsequent engraving. Prior to the scanning of page components, the operator uses a tablet in order to demark both cropping locations and to specify the locations of the components on the final page image. The scanned components are then assembled and coded by a single software process. The encoding process reduces the data storage requirement by a factor of two without any apparent loss of quality. Data is retrieved from disk storage, buffered, decoded, transmitted to a special formatter, and used to drive the Helio_Klischograph, which engraves the cylinder. Completely arbitrary imposition (the arrangement of pages on the gravure cylinder) is accomplished at the time of engraving. Provision is made for the arbitrary intermixture of comuter processed pages and conventional engraving by means of Cronapaques mounted on the companion scanning machine. Use of computer essentially eleminates the photographic steps required in the preparation of input copy as well as permits retouching and makeover due to more precise tone_scale control. In addition the engraving set_up time is significantly reduced due to the diminishing use of Cronapaques.

However, the present system does not provide any color processing through computer power, thus the quality of the printed image suffers from the limitations of the present technology. The work presented in this dissertation is aimed at channeling the computational power and reliability of a modern generation digital processor, to obtain enhancement in quality of color reproduction.

1.4. The Problem

The basic requirement of the data processing system for rotogravure color printing is that it be capable of receiving input from some type of color scanner, recording the data on mass-storage media (often magnetic disks), and eventually generating real_time data streams, while the Helio_Klischograph engraves multiple cylinders as required for four_color printing. Input images are typically scanned in as Red, Green and Blue separations. The engraving system requires that the input color_vector be mapped into a corresponding vector in the ink_density space. Since the inks are not perfectly transparent and also have complex mixing properties, this mapping turns out to be of a non_linear nature.

A simplistic approach for the engraving phase data processing could be to precompute the output images corresponding to Yellow, Cyan, Magenta and Black inks and have those stored on the disk, so that when actual engraving commences, the appropriate image could be retrieved and be used to generate the real_time data streams for engraving. But, this approach has a serious drawback. The input color_space being a 3_dimensional space, the requirement of storage for a colored image in the input domain would be atleast three times as much as a monochrome image (unless data is compressed using some clever coding scheme). Worse still, if the images were to be stored in the output domain, the output color_space being a 4_dimensional space, the storage requirement for the same image would now be four times as much. A typical monochrome image of a cylinder requires approximately 100 million bytes of data. Hence, storage efficiency would be seriously jeopardized if this approach were to be taken. An alternative approach is to compute the output images in real_time during the engraving_phase.

Like many facsimile systems, the Helio, once started, must be supplied with a real_time data_stream. The actual engraving takes approximately one hour with a data rate of 86.4 kilobytes/second. This imposes an upper bound of 11.5 microsecs on the processing time for the computation of ink_densities from every R_G_B sample corresponding to a "pel"(picture_element).

1.5. The Solution

The real_time data processing requirement of engraving phase demands that a high performance digital processor must be employed to compute the ink_density data_streams needed to drive the Helio. In the interest of flexibility, it was decided to use a programmable machine so that Ink Correction algorithm, being in software, could be altered in future, if necessary. Also, since the Ink Correction look_up tables are to be derived on the basis of experimental matching (described in section 2.7), it was foreseen that these data sets would be massaged over a number of trials for obtaining higher accuracies. These two basic requirements ruled out the use of a random_logic type hardwired processor. The application environment appeared to be quite suitable for commercially available microprocessor_type architecture and suggested a leading off_the_shelf microprocessor as a good fit. Consequently, a variety of contemporary high_end microprocessors were investigated, the choice eventually converging to Intel's 8086, mainly because of its powerful arithmetic instructions and efficient table_lookup capablities. However, the investigation showed that the thruput of a single MPU was much too low for the real_time operation. Presence of some parellelism in the Ink Correction algorithm suggested multiple_processor architecture to suppport parellel processing on independent data elements. Pipelined computation through sequential sharing of tasks on multiple_MPU as well as serial_parellel grid organisation of MPUs were also considered . However, the serial_parellel grid organisation of MPUs, necessary for generating the required thruput, turned out to be highly arbitrary as well as extremely inefficient and uneconomical. This was to be expected because the commercial microprocessors are not designed for such signal_processing applications, their architectures being optimized for entirely different considerations. The basic deficiencies (from the viewpoint of this application) in the architecture of these microprocessors were found to be the following;

1. Too few data registers, entailing high overhead of memory references.
2. I/O via accumulator, entailing high I/O overhead.
3. Highly sequential operation (zero or little concurrent processing).

The result of this investigation made it evident that a specially architectured processor, suitable for high_speed computation, was required to be designed to meet the needs of this application. The desirable characteristics of such a machine are as follows;

1. Special architecture supporting high degree of concurrent processing.
2. Bipolar TTL technology supporting high_speed system clock.
3. Powerful arithmetic processing capablities.
4. Large number of data registers.
5. Microprogrammable, to provide optimum performance as well as flexibility.
6. Immediate data operand from instruction_stream, suitable for co_efficients.
7. RAM for holding Ink_Correction Tables.
8. Powerful/fast Microprogram sequencer.
9. Intelligent system manager, for house_keeping functions.
10. Communication with a large host Computer such as PDP-11, to facilitate software development.

To meet these objectives, the Ink_Correction Module(ICM) was designed and implemented.

CHAPTER 2

Color-Correction by Computer Power

The current techniques used by the printing industry, to obtain high quality color reproduction, are difficult and expensive. The degradation in quality of printed image is largely due to the loss of color fidelity with respect to the original copy (assuming that the colors in the original copy were satisfactory). This loss occurs due to a variety of reasons, predominant among which are the physical characteristics of the printing inks, which impose a bound on the range of colors that can be produced by mixing these inks. The inks mix physically in a complex manner (an accurate model for this behaviour is still a topic of current research) causing the mapping of input colors into output ink densities to be a complicated problem. Also, original image itself may have colors which are less than satisfactory, making preprocessing of colors highly desirable. Hence, to enhance the quality of color reproduction, color_correction must be done at appropriate stages of image processing. The current technology attempts to do some color_correction (described in section 2.5), but can only do so very imprecisely owing to the limitations of the tools used. Undoubtedly, if these tools could be changed to more powerful and sophisticated ones, corrections would be more accurate resulting in better quality of color reproduction. A digital computer as a tool for color_correction, would yield tremendous enhancement in quality besides many other advantages.

2.1. Notion of Correct Color Rendition

If the original copy were perfect and if the system could reproduce all possible original colors, then exact reproduction would be preferred in most cases. Neither condition is true. Originals may not be perfect, and, more significantly, the gamut of the printing inks is much less than the gamut of possible input colors. Also, once the translation between input colors (of a perfect copy) and attainable output colors is established, actually achieving it is the most important and most difficult goal of any color systems [12]. Further, in some cases, it may be desirable to change the original color of an object, to accentuate another object in the image, or for special effects. It seems then that the notion of correct color is essentially tied to aesthetic choices. The color processing system, therefore, must provide the capability to change colors in an image arbitrarily, so that an operator, viewing it on a TV display, can interactively determine the correct color. We are then essentially faced with two types of color correction. One type of correction is concerned with the aesthetic choices involved in deciding what the output colors should be, either for perfect copy to be reproduced as well as possible, or imperfect copy which is to be altered in some way. The other type of correction is concerned with providing compensation for the inadequancies of printing inks in order to achieve the desired colors. As suggested by Schreiber [12], an important desirable feature of the color_processing system would be to deal with these two aspects of color_correction entirely independently of each other.

2.2. Color Correction via Interactive Editing

Schreiber et. al. have proposed a TV_based interactive color correction system to produce high quality printed images [12]. The central element in this color_correction system is the TV, since all the data from the input copy are converted to additive ICI values for the TV display, and then reconverted into ink densities for output. The part of the system which allows instantaneous viewing of the input copy or the output press print is known as the Color Translation Module. The CTM is a hardwired digital image processing system which has an interface to the host computer (PDP 11/34) as well as the TV display. With some help from computer an input copy from mass storage (if the input has been scanned in earlier and stored there) can be displayed on the TV via the Color Translation Module. A special operator's console attached to the CTM provides extensive color editing facilities. For example, the operator could change the color balance of the image in terms of either hue or saturation or both, separately for the highlights, midtones and/or the shadows. This helps to get rid of any unwanted color bias that the image may have acquired earlier. Alternatively, a color bias may be introduced for special effects. CTM also provides for selective color correction, in which, the seven principle colors Magenta, Blue, Cyan, Green, Yellow, Orange and Red can be selectively enhanced or attenuated in terms of either hue or saturation or both. For special color correction, CTM provides a color domain filter to be defined by the operator, so that an object in the image may be isolated on the basis of its hue and saturation and its color manipulated as desired. Since the gamut of the TV is larger than the gamut of printing inks, it is possibe that an operator may attempt to set a color which is not printable. But when that is done, CTM gives a warning.

An essential component of CTM is the gradation module, which provides the capability of tone scale transformation. Since tone_scale transformation is of primary importance to picture quality, special attention is paid to the control of this attribute. Any aesthetic tone_scale manipulations can be done by the operator interactively, separately for the highlights, the midtones and the shadows. The tone_scale compensations required to account for the physical characteristics of the particular choices of paper and ink, as well as, slow changes in stylus characteristics, are provided dynamically by the computer. The result of these tone_scale transformations is that the desired overall transfer characteristic is obtained.

Thus, the output from the Color Translation Module is the final image which has been compensated for its original defficiencies to the satisfaction of the operator. This image is then stored on the disk in a coded form, to be retrieved during the engraving phase. The image processing associated with mapping of colors from this form to printing ink densities is carried out on the engraving_phase data processing system described in chapter 3.

2.3. Choice of Color-Space

A colored image may be viewed as the combination of three images, one correspondeng to each primary color. Typically, in color_processing systems employing electronic scanners, these triple_images are generated by scanning in Red, Green and Blue separations. An image, thus, may be acquired in the R_G_B color_space. Historically, much of color science has been studied using R_G_B color space, because spectral red_green_blue components of natural white light provided a convenient primary system. Hence, R_G_B primary system became the standard basis for representing color images. Another reason for the popularity of R_G_B system is due to TV technology, since TV tubes have phosphors which also emit light in the red_green_blue spectral regions.

However, from the view point of color processing or storage, R_G_B color space is not an appropriate choice. The reason for this is that each of red, green, blue primary has some luminance associated with it, which implies that the colors in the image (color in strict sense meaning chromaticity as defined by hue and saturation only) could not be changed without affecting brightness (luminance). Such a restriction not only makes color manipulation extremely complicated but also destroys any possibility of data_compression. Hence, an essential requirement of the color_space to be used for color_correction as well as storage is that it must be a luminance_chrominance space. Fortunately, it is possible to create such an arbitrary system by performing a simple rotation of axes on the R_G_B color space. The result of such a rotation is that the luminance of a stimulus is confined to one_axis whereas its chromaticity is defined in the plane perpendicular to luminance_axis. NTSC Television broadcasting system essentially operates on this principle. A convenient luminance_chrominance space, suitable for color processing, may be defined by the following transformation matrix;

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1.40625 & 0 \\ 1 & -0.671875 & -0.21875 \\ 1 & 0 & 1.84375 \end{bmatrix} \begin{bmatrix} L \\ C1 \\ C2 \end{bmatrix}$$

Separating the luminance from the chromaticity of a stimulus yields a lot of benefit. For example, in order to change the chromaticity only of a stimulus, the 2_dimensional C1_C2 vector may be manipulated, leaving the luminance component completely unchanged. Obviously, such a representation system lends itself well to color processing. Another big advantage of this representation is that it enables data compression in several ways. It has been shown that the luminance component can be coded with fewer than 8_bits normally required for acceptable picture quality [8]. However, the main concern of the color system is to obtain some data compression on the chromaticity component, since the luminance component is just the equivalent of the grey_scale image of a monochrome system and does not demand additional bandwidth relative to a monochrome system. To obtain data_compression on chromaticity components, advantage is taken of the well known fact that the human eye has a much lower spatial acuity to chrominance information than luminance. Hence, chromaticity components may be coded at sub_resolution (typically one_fourth that of luminance component) without any apparent loss in picture quality. If chromaticity components are coded at one_fourth subresolution, only 50% more bandwidth is required to accomodate a color page as compared to a monochrome page. In terms of storage, this data compression yields very significant economy, since normally, four times as much storage would be required for the color page as that for the monochrome page.

2.4. Printing of Color Images

There are basically two different ways to produce color by mixing primaries. In "additive" system, as the name implies, the resultant light energy at each wavelength is the sum of that of the components. For example, the TV tube works on this principle. In subtractive mixing, which might more appropriately be called multiplicative, the transmittance at each wavelength (of a transparency, for example) is the product of the transmittance of the several components. Each component may be thought to subtract out certain colors. Photographic transparencies are almost perfectly subtractive, halftone prints are hybrids (additive where the dots are side by side, subtractive when the dots are superimposed), while gravure printing is more nearly subtractive, since the dots are not so well defined. Since a wide gamut of colors can be made by additive mixing of bright red, green and blue lights, it is obvious that an equal gamut could be produced subtractively if each ink or dye controlled the amount of reflected light in only one of the three wavelength bands- blue, green or red. An ink which was perfectly transparent to the green and red bands, but, when laid down in varying densities, absorbed more or less blue, would look colorless at low density and saturated yellow at high density. Typical yellow inks behave very nearly this way. An ink which was completely transparent to the blue and green bands, but absorbed more or less red, would look colorless at low density and bright blue green at high density. The technical name for such an ink is cyan. The third ink should transmit red and blue perfectly and absorb only more or less of green. It should range from colorless to bright purple, its technical name being Magenta. The real inks, however, do absorb in the undesired bands also and thus cause limitations which are discussed in the section 2.5.

Yellow, Cyan and Magenta are the three primary color inks invariably used in the printing industry. Even though, in theory, it should be possible to reproduce any color image using only the three inks mentioned above, in practice, however, a Black printer is almost always used additionally for the following reasons;

1. Since printing inks absorb a different proportion of light at each wavelength, specifying their density requires some convention. A useful measure is Equivalent Neutral Density(END) which is the density of the neutral Grey which results when the amount of color ink in question is mixed with appropriate amounts of the other two. This END concept makes it clear that in the case of a non_neutral color where all three inks are to be used, equivalent results should be obtainable by reducing the three color inks by a certain amount and substituting some amount of black. Naturally, the reduction can not be more than the lowest END of the three inks. This procedure, called Under Color Removal (UCR) is used because it

saves expensive color inks and also because it minimizes the effect of misregistration by putting more of the imagery into one separation.

2. Even without UCR, picture quality is improved with the use of a small amount of black imagery, called a "skeleton black printer". This is because maximum amounts of the three inks, as formulated to give good performance in highlights and midtones, can not achieve high enough density in shadows.

Different rasters (arrangements of cells) are normally used in four_color printing to minimize moire' effects [2,12]. The black printer (key) and magenta are engraved in an elongated raster which consists of 95 cells per inch around the circumference of the cylinder with a pitch of 253.1 lines per inch. A compressed raster consisting of 128 cells per inch circumferentially, with a line pitch of 166.5 per inch, is used for cyan and yellow. As there are four pages around the cylinder, the elongated raster consists of 1024 cells per line for each page and is thus a natural choice for the disk storage format for all the four separations. Yellow and cyan pages are therefore required to be interpolated during engraving.

2.5. Limitations of Real Inks

Even were the dot structure obliterated, gravure would still not be perfectly subtractive because the inks are not completely transparent and because there are physical interactions among the inks as they are laid down. For this reason the precise color actually produced by given amounts of the real inks can only be determined experimentally. A very important difference between additive and subtractive mixing is that the result of an additive mixture depends only on the appearance of the primaries. In subtractive mixing, however, two primaries which appear identical under white light, may mix very differently. This particular property of subtractive mixing accounts for the fact that look_alike printing inks may behave very differently on the press, thus underscoring the importance of empirical determination of the mixing results. It should also be noted that the illumination of a colored surface by colored light (by definition, "white" light is equal energy at all wavelengths) is subtractive. For example, colors which match under incandescent light may not match under fluorescent light.

Further, the real inks absorb light outside the optimum wavelength_band, thus imposing further limitations. For example, the real cyan inks, also called blue or process blue by printers, generally absorb quite a lot of green and some blue, in addition to red. This makes them look bluer and darker than true cyan. Similarly, real magenta inks absorb quite a lot of blue, making them look redder, plus some red, making them look darker. This unwanted absorptions of the real inks outside the optimum wavelength bands reduces the gamut of colors which can be reproduced by mixing them. In particular, bright saturated greens and bright saturated blues are generally not attainable. In addition, the need to achieve a neutral grey scale requires different contrasts of the three color images. If however, each of the printing ink densities was simply made equal to the corresponding original dye density, but adjusted in contrast so as to achieve a neutral Grey scale, very poor color reproduction would result. The color contrast and saturation would be greatly reduced. To lessen these effects a photographic process called masking is used as follows; When the magenta ink is printed in order to reproduce the green_light contrast of the original, a lower contrast second image also gets laid lown which modulates the blue light of the output image. This unwanted blue image can be neutralized at the cost of raising the overall contrast of blue light by reducing the density of the yellow ink, which is supposed to modulate the blue light, by an amount proportional to the contrast of the magenta ink. Since each of the three inks incorrectly absorbs in two other bands, it theoretically requires six masks to cancel all six unwanted absorptions, plus three more to undo the contrast-reduction effect of the masks on the main images. Because of the high cost of masking process, the practice in the current technology is to use only two or three masks for color correction, while contrast control is often achieved by varying development [12].

Complicated as this procedure seems, it still is not sufficient if best reproduction is desired throughout the tone scale. Seperate masking in highlights and shadows may also be desirable. Using the photographic masking techniques, only operators with very long experience and a high degree of skill are able to cope with the complexity of the process. Undoubtedly, the absorption of printing inks in the other two bands can be undone much more precisely through the powerful arithmetic capabilities of a digital computer.

2.6. A Strategy for Ink Correction

The computation of ink densities essentially involves mapping a 3_dimensional RGB (Red, Green, Blue) vector into a 4_dimensional YCMK (Yellow, Cyan, Magenta, Black or "Key") vector. In principle, this mapping can be achieved directly by using a lookup table. However, if 8_bit resolution is used for the input R_G_B vector as well as the output Y_C_M_K vector (Earlier experiments have shown that 8-bit resolution is the minimum for acceptable picture quality), the lookup table would be larger than 64 megabytes in size. Such a lookup table is clearly not desirable. A more pragmatic approach, therefore, would be to use a much smaller look_up table to store corrections at much reduced resolution and then use some clever interpolation scheme. Because the printing inks exhibit complex mixing behaviour described earlier, the corrections would have to be determined empirically by actually printing ink patches in incremental steps of density and matching them to TV R_G_B values. The number of colors for which exact corrections are available depends upon the size of the lookup table. This lookup table, called Ink Correction Table, may be viewed as a 3_dimensional grid, as shown in figure 1 (b), in which the exact corrections are available at the intersection points as a result of matching. But since the cells in the grid are large relative to actual resolution, some interpolation scheme must be used for all intermediate values. The accuracy to which the intermediate values may be corrected depends upon the non_linearity of the correction space and the type of interpolation function used. A reasonable approximation to the intermediate values of corrections may be computed using linear interpolation, which, although computationally non_trivial, does not involve any complicated arithmetic. In order to keep the color matching experiments within reasonable limits and the size of the Ink Correction Table manageable, it was decided that a table with 512 entries would be used. This allows a resolution of eight levels in each dimension of the correction space. The Ink Correction Table, therefore, has to be accessed by a 9-bit address (3-bits in each dimension), and has exact value of corrections stored for 512 colors corresponding to the same number of ink patches.

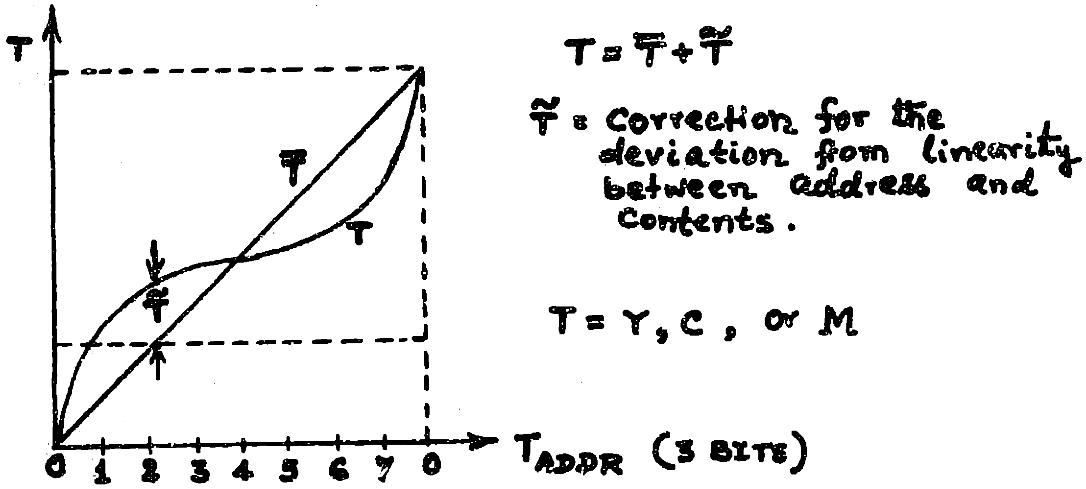


FIGURE 1(a) RELATIONSHIP BETWEEN LINEAR AND NON-LINEAR COLOR CORRECTION

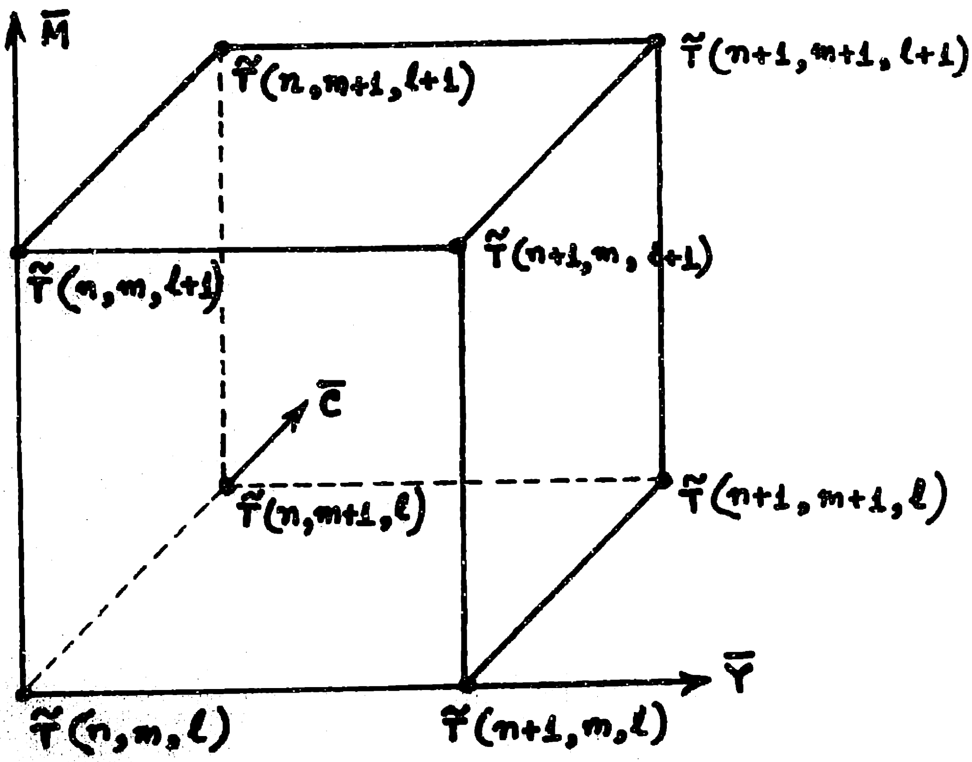


FIGURE 1(b) 3-DIMENSIONAL INK CORRECTION SPACE

2.7. Empirical Determination of ICTs

It is important to minimize the size of the entries in the ICTs, so that data sizes in ICTs are reduced saving storage space as well as simplify computation. In order to do this, input variables are operated upon by an algorithm which results in nearly linear relationship between the table addresses and its contents. Then, corrections to the output are stored in the table rather than the output itself. Since the corrections are much smaller relatively, the table size is reduced for the same order of accuracy. In addition if the axes of the input color space were rotated so as to nearly align with that of the output color space, many of the entries would become very small. Figure 2 depicts the algorithmic process for minimizing the table entries, the output variables thus generated are approximations $\bar{Y}$, $\bar{C}$, $\bar{M}$ to the output ink densities Y, C, M.

The matching experiments required for empirical determination of table entries are described by Schreiber and Berberian [3,4,5,6,7]. In these experiments, a book of ink patches, printed in 8 equidistant steps of densities in all four colors, is used to match the ICI values of the TV colors. The book does not contain the combination with the Black at density=1.0, which obviously does not require matching. Hence, the book contains 56 pages of 64 patches each. However, because of under_color_removal effect, a number (upto 16) of these ink patches will match each triad of ICI values.

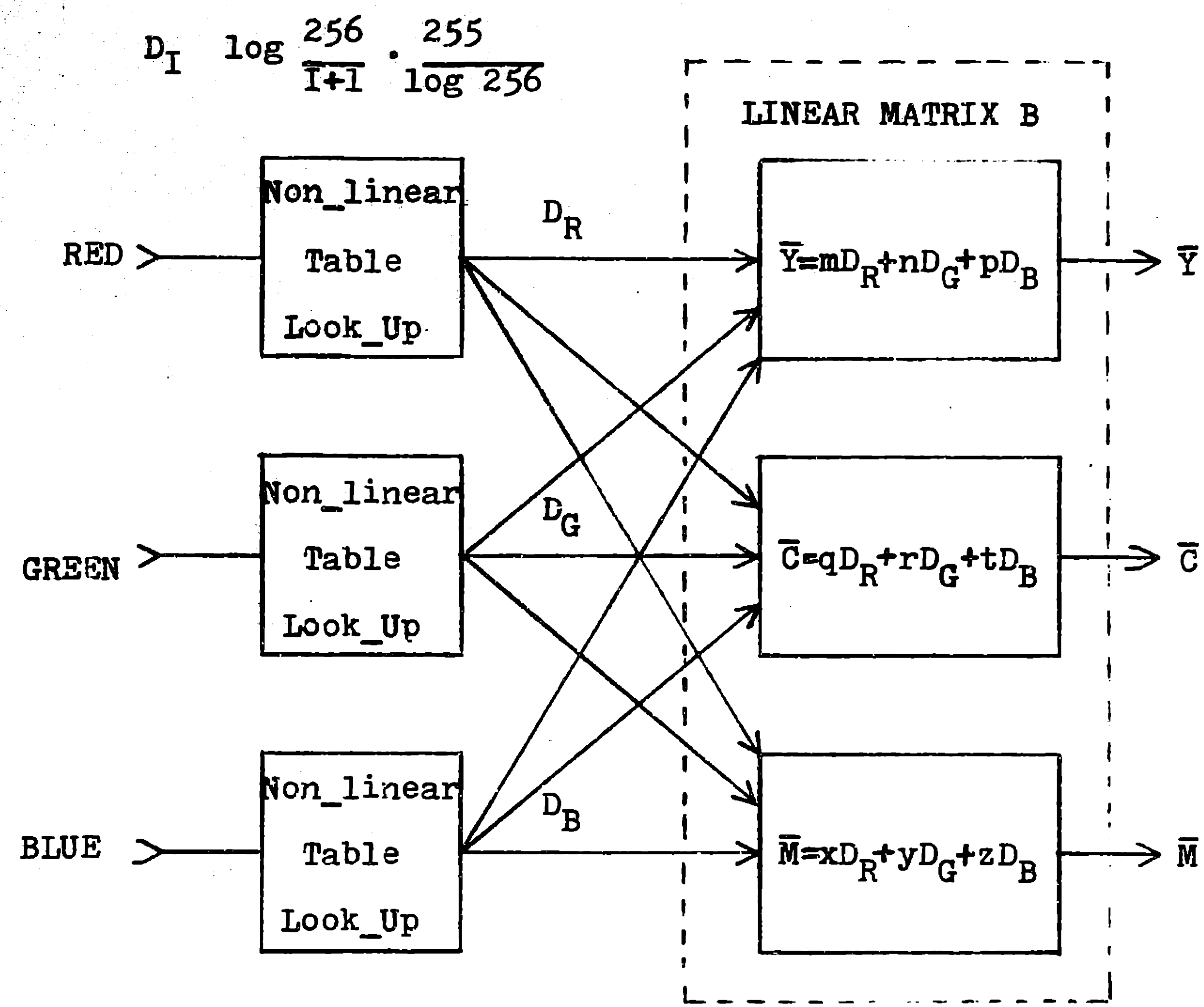


Figure 2 Algorithm for Minimization of Errors

But, for a given degree of UCR, only 512 patches will actually correspond to the matched data. Once the matching is done, $\bar{Y}\_\bar{C}\_\bar{M}$ may be computed corresponding to ICI vaues from TV, thus yielding the base value of correction for every point in the correction space (i.e.every address of the Ink Correction Table). The other entries of ICT, viz. first and second forward differences as defined in section 2.8, are derived from the base corrections. The reason for precomputing these differences, is simply a space_time trade_off.

2.8. An Ink Correction Algorithm

Figure 3 depicts the flow graph for the Ink Correction algorithm. The Red_Green_Blue values corresponding to a picture element are to be first converted to ideal dye densities by a non_linear operator. This non_linear mapping is logarithmic in nature because the relationship between transmittance and dye density, given by the Lambert_Beer Law [9], is as follows;

$$\tau(\lambda) = 10^{-d\,D(\lambda)}$$

Where $\tau(\lambda)$ is the transmittance function;

d, the dye density or concentration and

$D(\lambda)$, the spectral density function of a unit concentration of dye.

In the digital_processing domain, the actual operator used is given by the following equation;

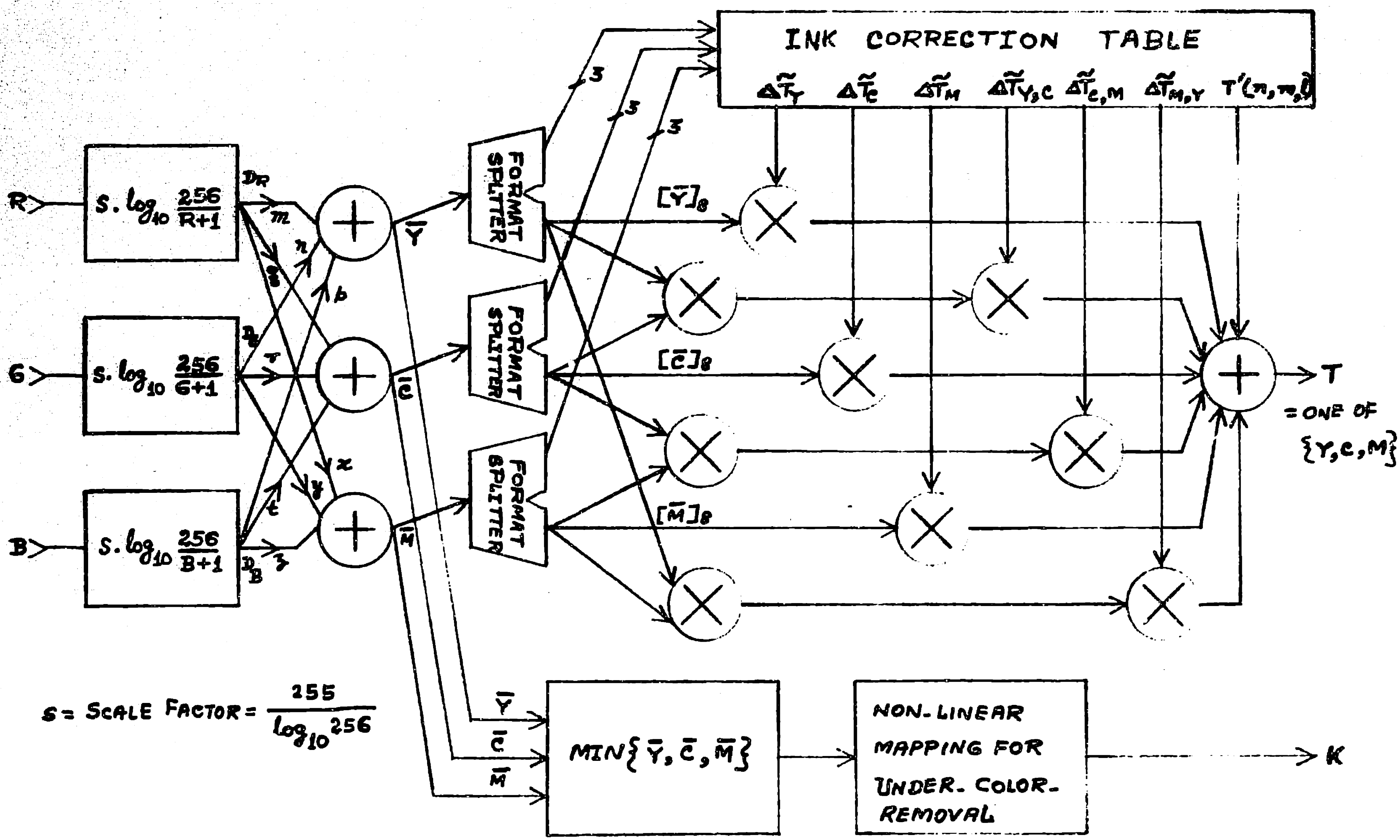


FIGURE 3 INK CORRECTION ALGORITHM - SIGNAL FLOW GRAPH

Given

$$0 \leq \{I, D_I\} \leq 255,$$

$$D_I = \left(\log_{10} \frac{256}{I+1}\right) \cdot \left(\frac{255}{\log_{10} 256}\right)$$

where

D_I, is the ideal block dye density corresponding to input I;

I, is one of R, G or B ICI values.

Both I and D_I are expressed as decimal equivalent of 8_bit binary words.

The spectral density functions of the ideal block dyes appear as shown in figure 4. A linear transformation must then be performed on the ideal dye densities D_R_D_G_D_B to yield approximations $\bar{Y}$_$\bar{C}$_$\bar{M}$ to the desired printing ink densities Y_C_M. The triad $\bar{Y}$_$\bar{C}$_$\bar{M}$ may also be viewed as non_ideal block dyes having spectral density functions which overlap in other two bands as shown in figure 5.

Mathematically, this transformation is as follows;

$$\begin{bmatrix} \bar{Y} \\ \bar{C} \\ \bar{M} \end{bmatrix} = \begin{bmatrix} & & \\ & B & \\ & & \end{bmatrix} \begin{bmatrix} D_R \\ D_G \\ D_B \end{bmatrix}$$

The effect of B-matrix may be viewed as to perform a rotation on the input vector D_R_D_G_D_B so that $\bar{Y}$_$\bar{C}$_$\bar{M}$ approximation is as nearly aligned to output Y_C_M as possible. The B-matrix emperically determined by Berberian [3], which

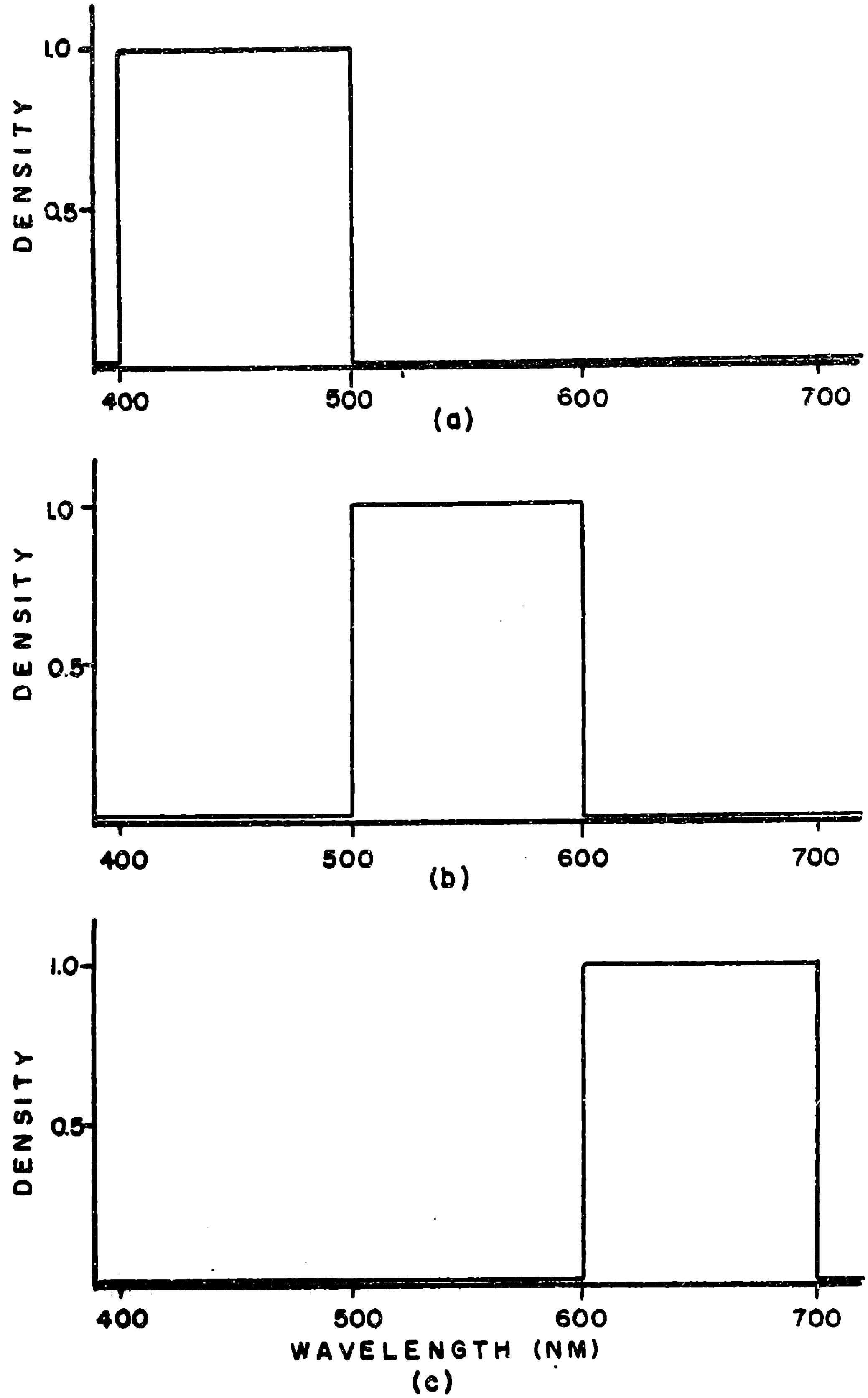


Figure 4- Spectral Density Functions of Ideal Block Dyes: (a) yellow, (b) magenta, (c) cyan.

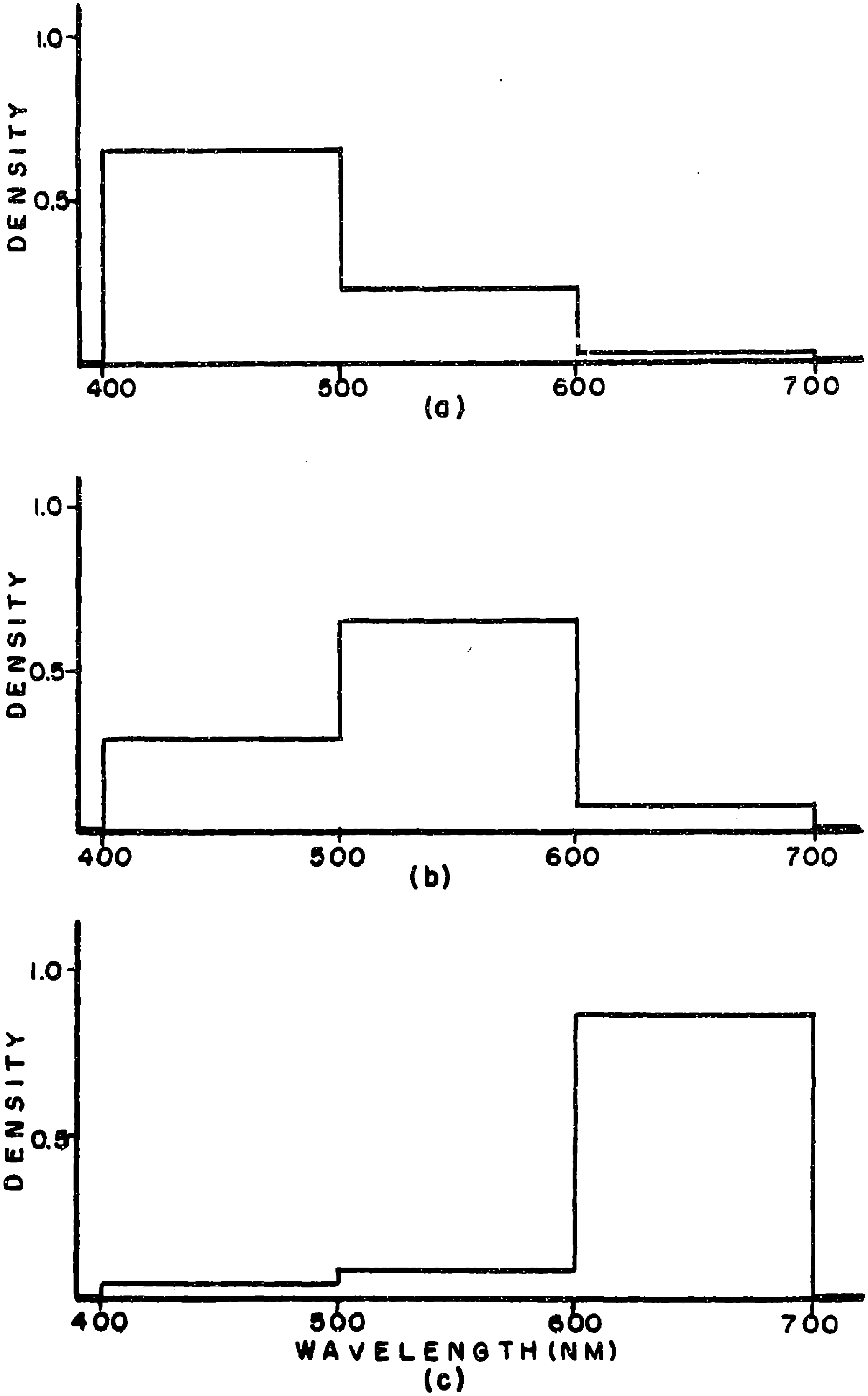


Figure 5 - Spectral Density Functions of 'Non-Ideal' Block Dyes: (a) yellow, (b) magenta, (c) cyan.

minimizes the maximum errors in this manner is as follows;

$$B' = \begin{bmatrix} 2.06 & -0.6309 & -0.1228 \\ -1.28 & 2.2 & -0.3 \\ 0.21 & -0.4046 & 0.6992 \end{bmatrix}$$

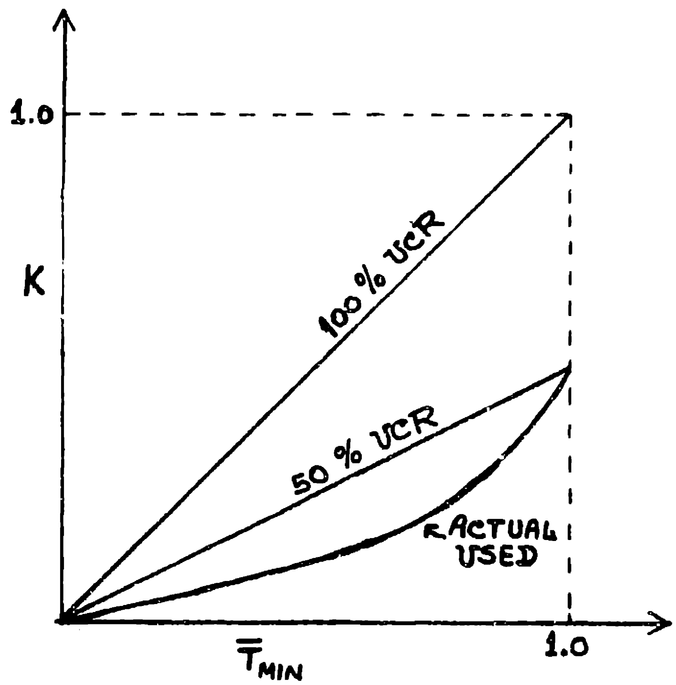
The same matrix, properly scaled and quantized for representation on 8-bit data path [10], appears as follows;

$$B = \begin{bmatrix} 1.8671875 & -0.5703125 & -0.109375 \\ -1.15625 & 1.9921875 & -0.2734375 \\ 0.1875 & 0.3671875 & 0.6328125 \end{bmatrix}$$

Once the approximations are known, the key (UCR component) may be determined as follows; The algorithm for computing the key (Black ink density) is essentially a two_step procedure. The first step involves determination of the least of $\bar{Y}$_$\bar{C}$_$\bar{M}$, because the UCR can never be larger than the least of these three as explained in section 2.4. In the second step, the key is computed as a non_linear function of the least value determined in step 1, as proposed by Schreiber [6].

For flexibility and convenience, a lookup table is used for the function depicted in figure 6.

To compute the color ink_densities Y_C_M, 3_dimensional linear interpolation must be carried out using the Ink Correction Table. The corrections are stored in ICT in a rather clever manner. This method, proposed by Schreiber, is based



$\bar{T}_{MIN}$ = MINIMUM OF $\{\bar{Y}, \bar{C}, \bar{M}\}$

K = DENSITY OF BLACK INK

FIGURE 6 NON-LINEAR MAPPING FUNCTION (MUCR) FOR UNDER-COLOR-REMOVAL upon the apriori knowledge of the relationship between the linear (ideal) and non_linear(real) ink_corrections, shown in figure 1 (a). Since the deviations of the corrections from linearity is small compared to its absolute value, significant storage efficiency is obtained by storing the deviations from linearity rather than the value of correction itself. The absolute value of correction can then be generated simply by summing the deviation to the linear approximation, which, in this instance, is the input variable itself. Figure 1 (b) depicts one cell of the 3_dimensional grid normally associated with the ink_correction space. Since ICT has 512 entries, it may be viewed to be consisting of 512 such cells, 8 in each direction of $\bar{Y}$, $\bar{C}$ & $\bar{M}$. Further, since the ink_density space is highly non_linear as well as assymmetrical,

three different Ink_Correction tables must be used for each Y, C & M output. Our objective is to compute Y_C_M, which are 8-bit binary numbers, as a function of the three inputs $\bar{Y}$_$\bar{C}$_$\bar{M}$, also 8_bit binary numbers. The three most_significant bits of $\bar{Y}$_$\bar{C}$_$\bar{M}$ readily point to the particular cell in which the object ink density value must lie. But, to obtain 8_bit precision along each dimension, the output variables Y_C_M must be interpolated from the 5 lower_order bits of inputs $\bar{Y}$_$\bar{C}$_$\bar{M}$ (within the cell). Each cell thus, may be viewed as being divided into 32-parts along each axis. Since, the values of base corrections $\tilde{\bar{T}}$ (one of $\tilde{\bar{Y}}$_$\tilde{\bar{C}}$_$\tilde{\bar{M}}$) are known (from ICT) for all 8 corner-points of the cubic cell, it is possible to linearly interpolate within the cell using 5 lower order bits from each of $\bar{Y}$, $\bar{C}$, $\bar{M}$ to obtain the 8_bit value of the output ink densities. Buckley has presented a detailed discussion on 3_dimensional linear interpolation [11].

Mathematically, 3_dimensional linear interpolation may be viewed as the computation of Taylor's series expansion on input variables $[\bar{Y}]_8$,$[\bar{C}]_8$,$[\bar{M}]_8$ (which are the 5 lower order bits of $\bar{Y}$_$\bar{C}$_$\bar{M}$) using pre_determined co_efficients (from Ink Correction Table) according to the following equations;

$$Y = \bar{Y} + \tilde{Y}(n,m,l) + [\bar{Y}]_8 \Delta\tilde{Y}_Y + [\bar{C}]_8 \Delta\tilde{Y}_C + [\bar{M}]_8 \Delta\tilde{Y}_M +$$

$$[\bar{Y}]_8 [\bar{C}]_8 \Delta\tilde{Y}_{Y,C} + [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{Y}_{C,M} +$$

$$[\bar{M}]_8 [\bar{Y}]_8 \Delta\tilde{Y}_{M,Y} + [\bar{Y}]_8 [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{Y}_{Y,C,M}$$

$$C = \bar{C} + \tilde{C}(n,m,l) + [\bar{Y}]_8 \Delta\tilde{C}_Y + [\bar{C}]_8 \Delta\tilde{C}_C + [\bar{M}]_8 \Delta\tilde{C}_M +$$

$$[\bar{Y}]_8 [\bar{C}]_8 \Delta\tilde{C}_{Y,C} + [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{C}_{C,M} +$$

$$[\bar{M}]_8 [\bar{Y}]_8 \Delta\tilde{C}_{M,Y} + [\bar{Y}]_8 [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{C}_{Y,C,M}$$

$$M = \bar{M} + \tilde{M}(n,m,l) + [\bar{Y}]_8 \Delta\tilde{M}_Y + [\bar{C}]_8 \Delta\tilde{M}_C + [\bar{M}]_8 \Delta\tilde{M}_M +$$

$$[\bar{Y}]_8 [\bar{C}]_8 \Delta\tilde{M}_{Y,C} + [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{M}_{C,M} +$$

$$[\bar{M}]_8 [\bar{Y}]_8 \Delta\tilde{M}_{M,Y} + [\bar{Y}]_8 [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{M}_{Y,C,M}$$

Where, referring to figure 1(b), the forward differences are defined as follows;

$$\Delta\tilde{Y}_Y = \tilde{Y}(n+1,m,l) - \tilde{Y}(n,m,l)$$

$$\Delta\tilde{Y}_C = \tilde{Y}(n,m+1,l) - \tilde{Y}(n,m,l)$$

$$\Delta\tilde{Y}_M = \tilde{Y}(n,m,l+1) - \tilde{Y}(n,m,l)$$

$$\Delta\tilde{C}_{Y} = \tilde{C}(n+1,m,l) - \tilde{C}(n,m,l)$$

$$\Delta\tilde{C}_{C} = \tilde{C}(n,m+1,l) - \tilde{C}(n,m,l)$$

$$\Delta\tilde{C}_{M} = \tilde{C}(n,m,l+1) - \tilde{C}(n,m,l)$$

$$\Delta\tilde{M}_{Y} = \tilde{M}(n+1,m,l) - \tilde{M}(n,m,l)$$

$$\Delta\tilde{M}_{C} = \tilde{M}(n,m+1,l) - \tilde{M}(n,m,l)$$

$$\Delta\tilde{M}_{M} = \tilde{M}(n,m,l+1) - \tilde{M}(n,m,l)$$

The second order differences are defined as follows;

$$\Delta\tilde{Y}_{Y,C} = \tilde{Y}(n+1,m+1,l) - \tilde{Y}(n+1,m,l) - \tilde{Y}(n,m+1,l) + \tilde{Y}(n,m,l)$$

$$\Delta\tilde{Y}_{C,M} = \tilde{Y}(n,m+1,l+1) - \tilde{Y}(n,m+1,l) - \tilde{Y}(n,m,l+1) + \tilde{Y}(n,m,l)$$

$$\Delta\tilde{Y}_{M,Y} = \tilde{Y}(n+1,m,l+1) - \tilde{Y}(n+1,m,l) - \tilde{Y}(n,m,l+1) + \tilde{Y}(n,m,l)$$

$$\Delta\tilde{C}_{Y,C} = \tilde{C}(n+1,m+1,l) - \tilde{C}(n+1,m,l) - \tilde{C}(n,m+1,l) + \tilde{C}(n,m,l)$$

$$\Delta\tilde{C}_{C,M} = \tilde{C}(n,m+1,l+1) - \tilde{C}(n,m+1,l) - \tilde{C}(n,m,l+1) + \tilde{C}(n,m,l)$$

$$\Delta\tilde{C}_{M,Y} = \tilde{C}(n+1,m,l+1) - \tilde{C}(n+1,m,l) - \tilde{C}(n,m,l+1) + \tilde{C}(n,m,l)$$

$$\Delta\tilde{M}_{Y,C} = \tilde{M}(n+1,m+1,l) - \tilde{M}(n+1,m,l) - \tilde{M}(n,m+1,l) + \tilde{M}(n,m,l)$$

$$\Delta\tilde{M}_{C,M} = \tilde{M}(n,m+1,l+1) - \tilde{M}(n,m+1,l) - \tilde{M}(n,m,l+1) + \tilde{M}(n,m,l)$$

$$\Delta\tilde{M}_{M,Y} = \tilde{M}(n+1,m,l+1) - \tilde{M}(n+1,m,l) - \tilde{M}(n,m,l+1) + \tilde{M}(n,m,l)$$

The third order difference is defined as follows;

$$\Delta\tilde{Y}_{Y,C,M} = \tilde{Y}(n+1,m+1,l+1) - \tilde{Y}(n,m+1,l+1) + \tilde{Y}(n+1,m,l) - \tilde{Y}(n,m,l) - \tilde{Y}(n+1,m+1,l) + \tilde{Y}(n,m+1,l) - \tilde{Y}(n+1,m,l+1) + \tilde{Y}(n,m,l+1)$$

$$\Delta\tilde{C}_{Y,C,M} = \tilde{C}(n+1,m+1,l+1) - \tilde{C}(n,m+1,l+1) + \tilde{C}(n+1,m,l) - \tilde{C}(n,m,l) - \tilde{C}(n+1,m+1,l) + \tilde{C}(n,m+1,l) - \tilde{C}(n+1,m,l+1) + \tilde{C}(n,m,l+1)$$

$$\Delta\tilde{M}_{Y,C,M} = \tilde{M}(n+1,m+1,l+1) - \tilde{M}(n,m+1,l+1) + \tilde{M}(n+1,m,l) - \tilde{M}(n,m,l) - \tilde{M}(n+1,m+1,l) + \tilde{M}(n,m+1,l) - \tilde{M}(n+1,m,l+1) + \tilde{M}(n,m,l+1)$$

In the above equations, ICT supplies the base_corrections $\tilde{T}(n,m,l)$ as well as all the first and second forward differences. The third order term, being insignificantly small, may be ignored. Also, equation (1) thru (3) above are valid only for 3_color system. Since, under_color_removal is used in the proposed system, the terms $\bar{Y}$, $\bar{C}$, $\bar{M}$ in the right_hand expressions may be replaced by Y', C', M' respectively such that ($\bar{Y}$ - Y'), ($\bar{C}$ - C'), ($\bar{M}$ - M') together constitute the key or the UCR_component. This implies that the entries in ICT take into account the effect of under_color_removal.

Appendix A presents Macro_ and Micro_Flow_charts for the Ink Correction algorithm in respect of the key and one of the color ink_densities.

CHAPTER 3

Engraving System Overview

Figure 7 outlines the organisation of the Helio_Engraving system. The coded disk_based data resulting from the Color Translation Process is retrieved by the Channel Processor and fed to Multi channel Data Converters. Data Converter decodes the input data for continuous_tone/lineart areas and generates elongated/compressed rasters . The Data Converter has a tone_scale memory, which provides company translation of the signal as well as compensation for stylus_wear. The data format for the color page is two luminance lines of 1024 bytes each followed by half as many bytes of chrominance information. The luminance is to be coded as one byte per cell for the contone area and two half bytes (nibbles) per cell for lineart. The chrominance information will consist of C1 in the low (even numbered) byte and C2 in the high (odd numbered) byte. The chrominance information will be stored at a sub_resolution of 4 cells per sample.

The Color Data Formatter (CDF) receives the data from MCDCs and expands the sub_resolution chrominance signals to full_resolution and outputs l_C1_C2 signals over 3 parallel channels as real_time data_streams. A hardwired module converts l_C1_C2 signal into R_G_B. ICM transforms the R_G_B signal into Y_C_M_K ink densities and outputs one of these at a time. The ICM output goes to another tone_scale memory, the function of which is to linearize the entire data_processing

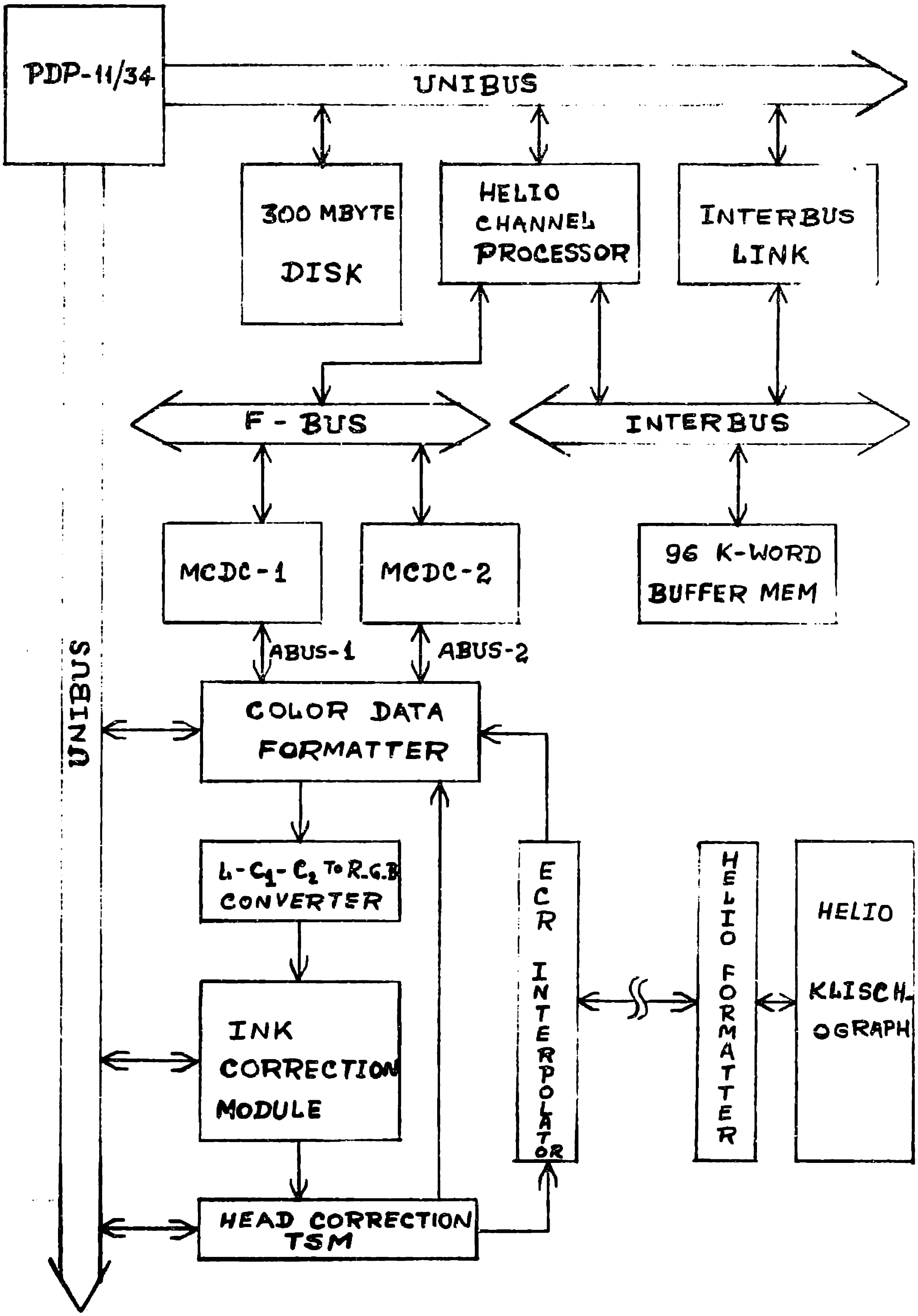


FIGURE 7 BLOCK DIAGRAM OF HELIO-ENGRAVING SYSTEM

chain. The output of TSM goes to ECR_Interpolator followed by the Helio Formatter. The ECR_Interpolator interpolates yellow and cyan pages from elongated to compressed raster. While, Helio Formatter provides the necessary interface to the Helio_Klischograph. ICM has a transparent mode which is used for monochrome separations as well as the initialization phase. Real_time processing by ICM is synchronous with the System Clock. ICM has a UNIBUS interface to communicate with the host PDP-11 Computer.

3.1. Interface Specification

Figure 8 illustrates the interface information in respect of the Ink Correction Module. All data inputs to ICM are sourced within the Color Data Formatter (CDF) and converted to R_G_B data_stream by a hardwired module. The control and status signals are provided directly by CDF. The system clock is echoed by CDF as per the time relationship shown in fig 9.

All signals are TTL. Data transfers through ICM are always synchronized with the system clock. Description of the input/output signals are as follows;

DATA1in:(Pin# MSB 16,14,12,10,8,6,4,2 LSB/CONN#J1)

During the initialization phase, this input carries the data for internal registers in the Helio Formatter. The ICM mode of operation is such that the output is same as the

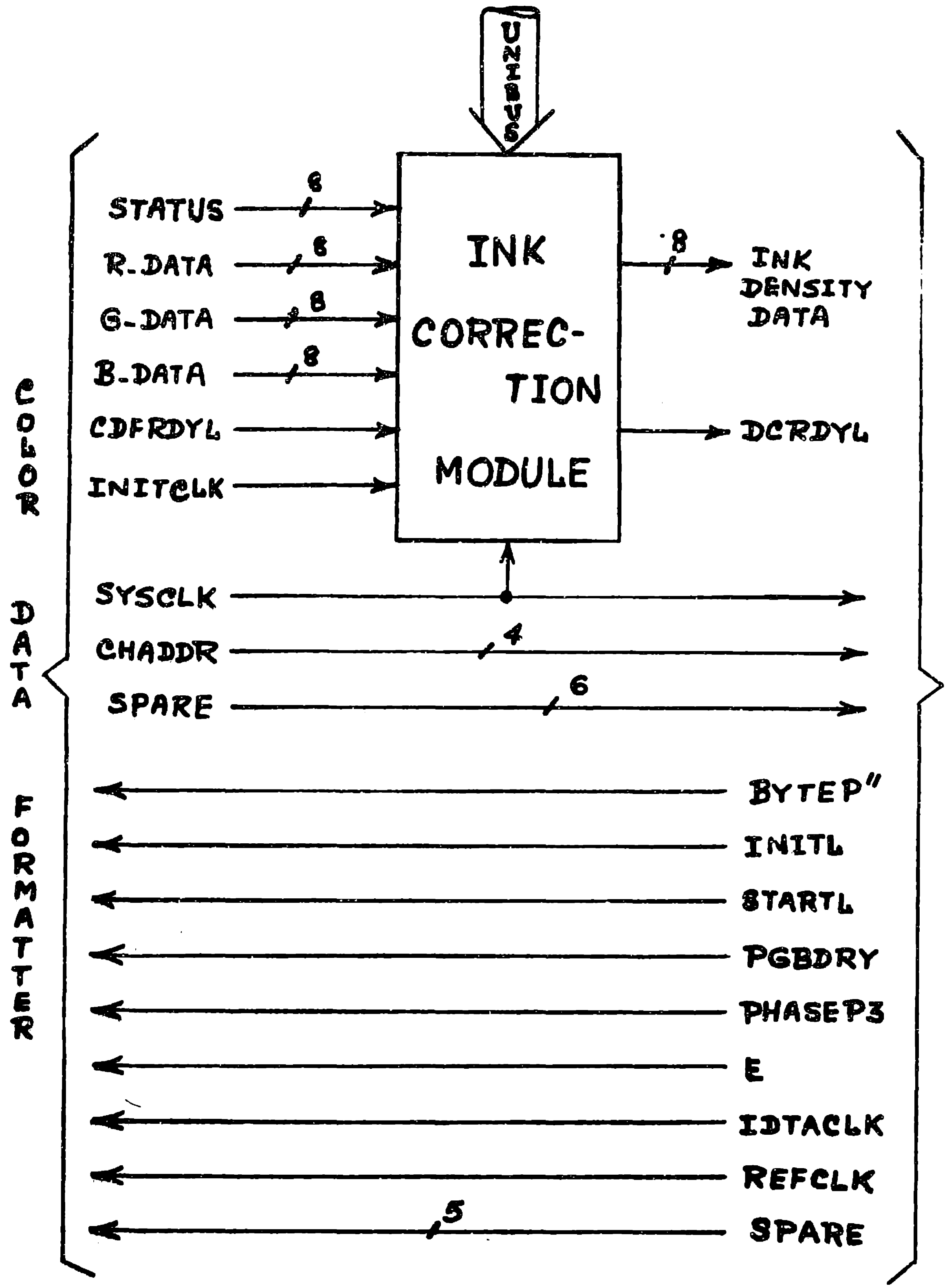


FIGURE 8 ICM INTERFACE

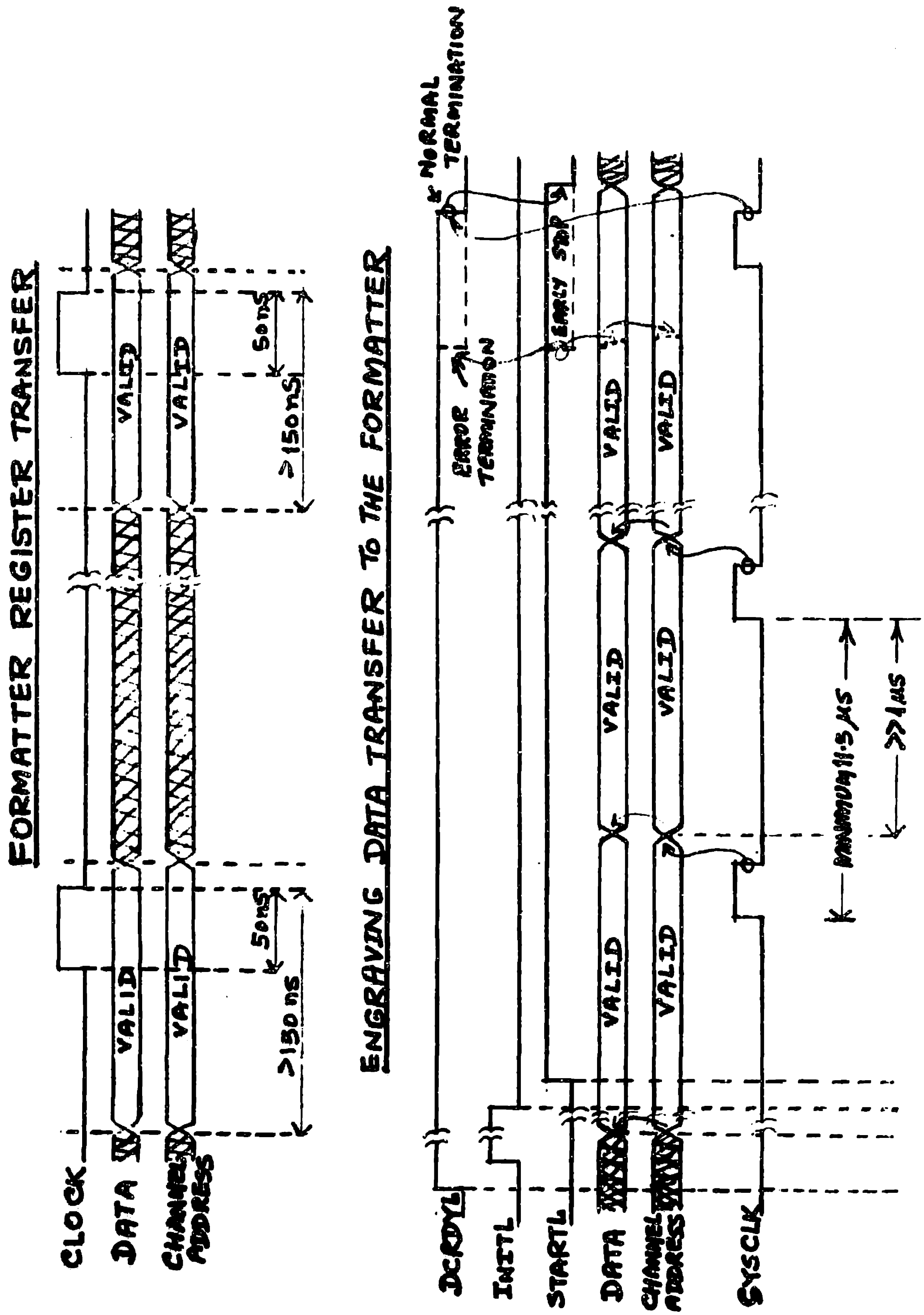


FIGURE 9 INTERFACE TIME DIAGRAM

input. CDF must ensure that this data_port is used during the initialization phase.

During the engraving phase, this input carries "Red" channel data

converted from 1-C1-C2 by the hardwired module. In other modes, this channel may be used for engraving monochrome images or color images directly from R_G_B separations type data.Status bits specify what ICM must do with this data.

DATA2in:(Pin# MSB 48,46,44,42,40,38,36,34 LSB/CONN#J3)

This input is active only during the engraving phase and only if the 'process' mode is operating. It carries the "GREEN" channel data of the R_G_B input.

DATA3in:(Pin# MSB 30,28,26,20,18,16 LSB/CONN#J3)

This is identical to DATA2in. except that it carries the "BLUE" channel signal of the R_G_B input.

DATAout:(Pin# MSB 16,14,12,10,8,6,4,2 LSB/CONN#J2)

This is the output from ICM.

CHADDRin: (Pin# MSB 28,26,24,22 LSB/CONN# J1)

Channel Address input. Since ICM does not use Channel Address signal, this is simply connected through, to the output.

CHADDRout: (Pin# MSB 28,26,24,22 LSB/CONN#J2)

Channel Address output. CA bits are relayed by ICM to the ECR Interpolator without any delay.

SYSCLKin: (Pin# 20 /CONN# J1)

The System Clock is echoed by the Color Data Formatter in response to the BYTEP" signal. Timings are shown in figure 9.

SYSCLKout: (Pin# 20 /CONN# J2)

The System Clock is relayed by ICM to the suceeding unit without any delay.

INITCLK: (Pin# 3 /CONN# J3)

During the PROCMODE, CDF generates INITCLK, a single pulse, immediately after the first data bytes are fetched and outputs to ICM. Effectively, this causes data bytes to be fed to ICM one clock ahead and thus permits pipelining of data through ICM, allowing a maximum of 11.5 microsec data-processing time for the ICM.

CDFRDYL: (Pin# 18 /CONN# J1)

This signal asserts that the CDF and the system preceding it are ready for the engraving phase.

DCRDYL: (Pin# 18 /CONN# J1)

CDFRDYL is ANDed with ICMRDY to produce the DCRDYL signal and thus ensures that the time required for any preparatory operation by ICM is provided and engraving does not begin until all functional modules are properly initialized.

STATUS:(Pin# MSB 13,12,11,10,9,8,7,6 LSB/CONN# J3)

Status bits are set in the CDF and are interpreted by ICM as shown in figure 10. The two LSBs represent the color presently being engraved. The next 4 bits selectively control whether engraving of any particular color should be suppressed. The two MSBs decide the operating mode.

Op-mode `00' is a real_time NOP mode. In this mode the engraving process is not active and therefore, the time may be used by ICM to execute any diagnostic procedures or any other house_keeping function, in commnication with the host_computer. During `01' op_mode, ICM simply outputs the data presented to it at DATA1in port. In this mode, ICM is transparent to the system and hence this mode should be used for engraving black or separations type images, as well as for initialization. `10' op_mode, is the real_time processing mode in which ICM executes the microprogrammed Ink_Correction algorithm on the input data, generating the value of ink_density.

SPAREin:(Pin# 30,32,34,36,38,40 /CONN# J1)

SPAREout:(Pin# 30,32,34,36,38,40 /CONN# J2)

UNIBUS INTERFACE: (DEC STD. Backplane A & B)

This provides the interface between the host PDP-11 Computer and the ICM. The Unibus is interfaced through the system manager of ICM, which is Intel's 8085 MPU based microcomputer. Extensive software is installed in the manager to provide a variety of functions.

The physical interconnection of various functional modules of the engraving_phase data processing system is depicted in figure 11. As shown, one 3_M cable, bypassing ICM, connects

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_ϕ	← STATUS BYTE
x	x	x	x	x	x	0	0	← COLOR MODE = BLACK
x	x	x	x	x	x	0	1	← COLOR MODE = YELLOW
x	x	x	x	x	x	1	0	← COLOR MODE = CYAN
x	x	x	x	x	x	1	1	← COLOR MODE = MAGENTA
x	x	x	x	x	1	x	x	← INHIBIT BLACK
x	x	x	x	1	x	x	x	← INHIBIT YELLOW
x	x	x	1	x	x	x	x	← INHIBIT CYAN
x	x	1	x	x	x	x	x	← INHIBIT MAGENTA
0	0	x	x	x	x	x	x	← OP. MODE = NOP
0	1	x	x	x	x	x	x	← OP. MODE = TRANSPARENT
1	0	x	x	x	x	x	x	← OP. MODE = PROCESS
1	1	x	x	x	x	x	x	← SPARE

FIGURE 10 EXPLANATION OF ICM STATUS BITS

Color Data Formatter directly to ECR Interpolator. This cable carries signals from ECR Interpolator viz. BYTEP", INITL, STARTL, PGBDRY, PHASE P3, E, IDTACLK, REFCLK and 5 SPARE lines, which have no interaction with ICM.

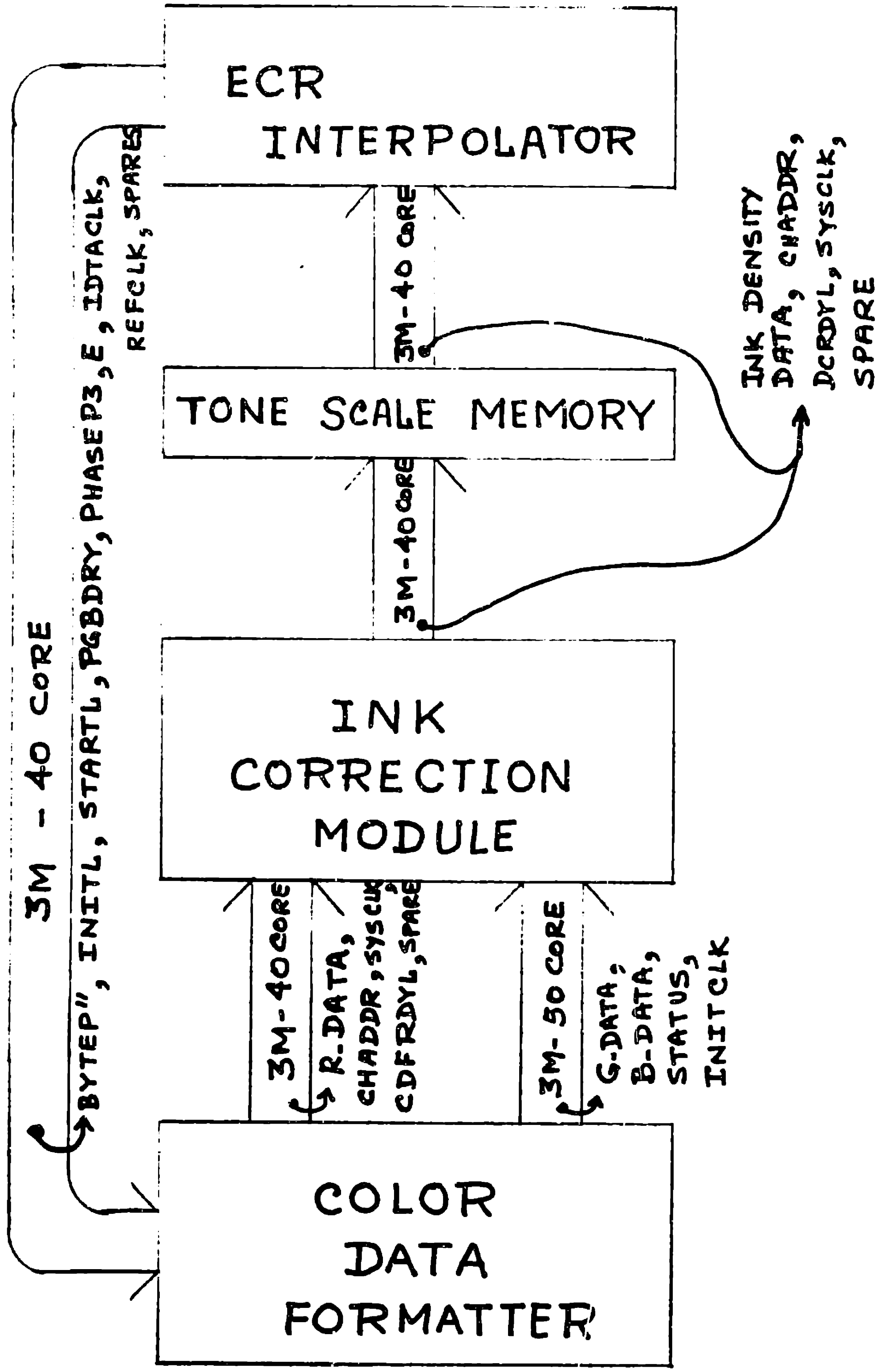


FIGURE 11 SYSTEM INTERCONNECTION OF ICM

CHAPTER 4

ICM System Architecture

The focus of ICM system architecture is a high_speed microprogrammable processor, intended to serve as the work_horse for all the real_time number_crunching associated with the Ink Correction algorithm. A Block diagram presented in Fig. 12 depicts the major components of the system which is broadly partitioned into two functionally independent sub_systems viz.(i) The Real_Time Processor(RTP) and (ii) The System Manager(MGR). Since, the real_time processor must not be burdened with the house_keeping functions which are also to be performed, an alternative must be provided to shoulder this responsibility. A microprocessor_based system manager is just perfect for the job. Accordingly, the manager was designed around an Intel 8085.

4.1. Hardware Architecture

The architecture of the real_time processor is based on the notion of supporting as much concurrent processing in a sequential machine as a reasonable amount of hardware would permit. Concurrency is derived through both parellel processing as well as by overlapping (pipelining). Bipolar Schottky TTL technology was the inevitable choice for implementation. The processor is organised around a high_speed dual_data_bus

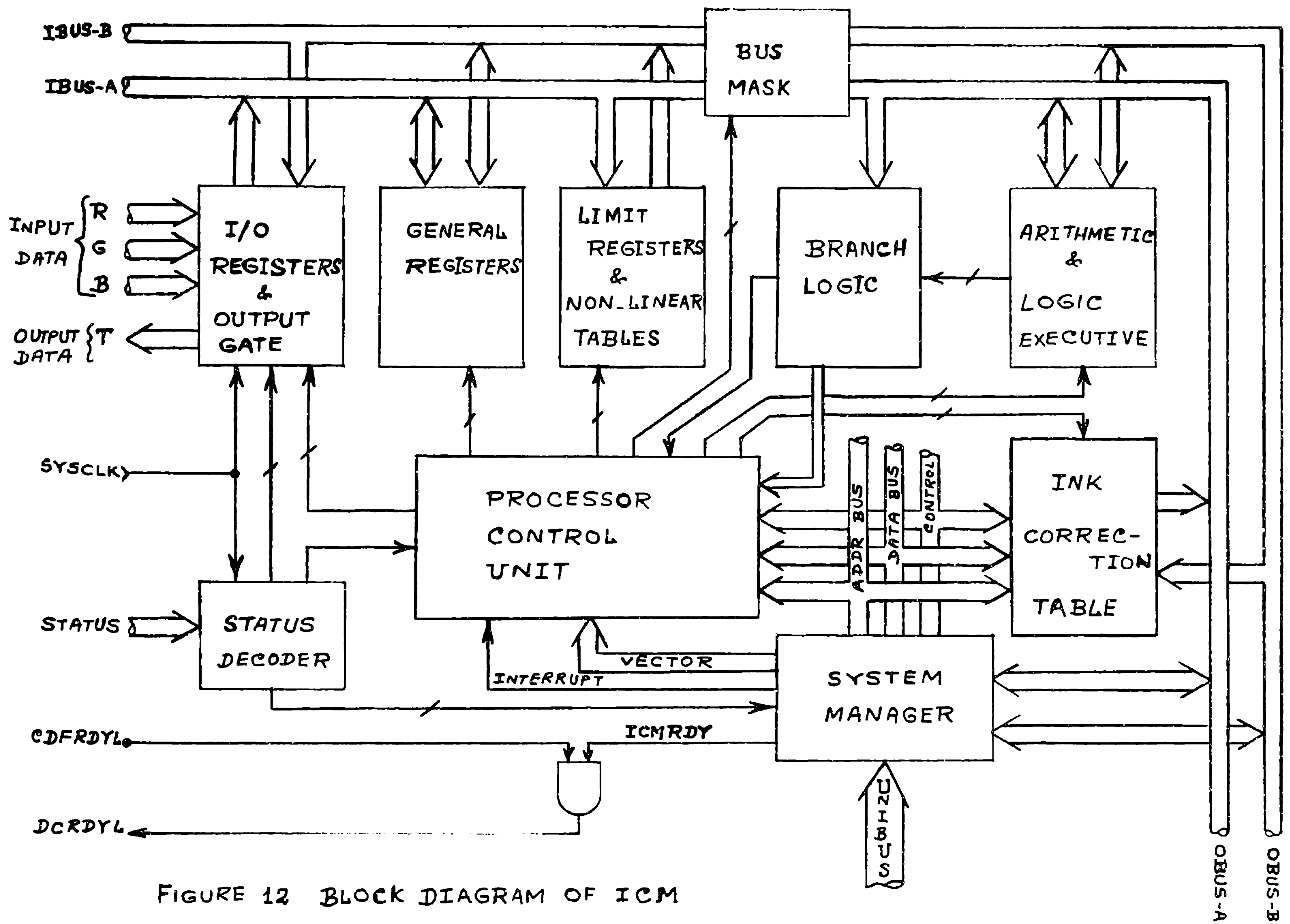


FIGURE 12 BLOCK DIAGRAM OF ICM

structure, which facilitates two bus_transactions during one microcycle, effectively nearly doubling the thruput. The greatest benefit from such a data_path is ofcourse derived when a double_operand transaction (which happens most of the time) occurs, since no overhead is incurred in fetching the operands sequentially which is characteristic of machines organised around a single data bus. However, the gain in efficiency would not be as much if a large number of single_operand transactions are enqued and both data_buses do not get loaded every microcycle owing to data_interdependency. From the viewpoint of real_time processing, this is perfectly acceptable because thruput is the prime consideration rather than efficiency. Further, in a number_crunching application, most of the time double_operand operations are executed. Fast and powerful arithmetic is provided by designing a multiple_element executive, which includes a high_speed Multiplier_Accumulator, a high_speed ALU and a shifter_rotator. Non_linear operations are mapped through use of lookup tables. Eight high speed dual port bidirectional registers are provided for holding partial results, whereas 2 sets of 3 unidirectional registers support I/O activity. A dedicated Data Interface matches all host system_timing requirements. For sequencing, a fast and powerful micropro-gram controller viz. AM2910 was chosen to operate at a system clock of 6.25 MHz thereby yielding a microinstruction cycle time of 160 nanoseconds. To obtain maximum concurrency, two_level pipelined microprogram control architecture is implemented as shown in figure 13. In a pipeline architecture, the fetch of the next micro_instruction is overlapped, while the current micro_instruction is still being executed. Two_level pipelined architecture actually entails three stages of pipelining, one in each signal path, and is thus known as Instruction_Address_Data based control architecture. It is called two_level pipelined because the programmer has to keep track of events that would occur two microinstructions later, and therefore, microprogramming in this architecture is much more difficult than other architectures. On the other hand, the two_level pipelined architecture is undoubtedly the fastest of all standard microprogram control architectures.

The Manager is primarily responsible for house_keeping functions such as power_up initialization, downloading microprogram and Ink_Correction table into RTP, communication with host processor (PDP-11), diagnostics and program development support. The manager is a microcomputer, based on Intel's 8085 MPU, with 4Kbytes each of program memory (PROM) and local data memory (RAM). Additionally, manager's MPU can access 2Kbytes each of RTP's microprogram memory and the Ink_Correction table (both modules are implemented with Bipolar RAM) which are mapped in the top half of manager's memory address space. RTP's microprogram memory and the Ink_Correction table are in some sense dual_port memories, which, in offline mode, can be read or written by the manager's MPU. But, during the real_time mode, these memory segments become read-only and are only accessible to RTP.

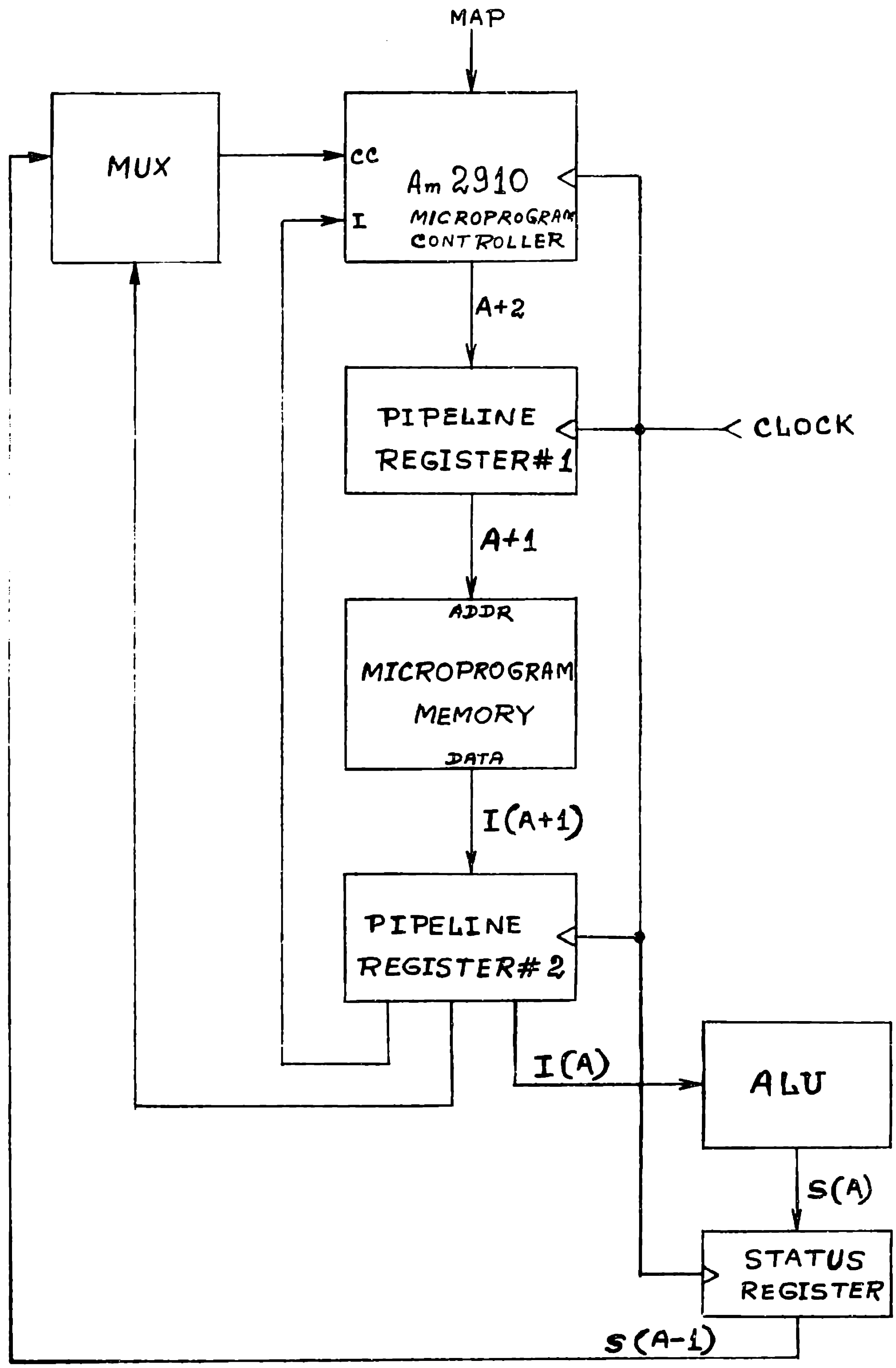


FIGURE 13 TWO-LEVEL PIPELINED MICROPROGRAM CONTROL ARCHITECTURE

Figure 14 depicts the memory address map of the manager.

Manager has a simple Interrupt structure as follows;

1. INTR: Hardware Interrupt or Restart 7 is used for recovery from hang_up in unused memory address space.
2. TRAP: is non_maskable and is used for recovery from hang_up by host processor.
3. RST7.5: is used for indicating color mode status changes. These mode bits are input from Color Data Formatter. Any change in color mode requires downloading of appropriate microprogram and ICT.
4. RST6.5: is used to implement the Unibus Interface. Every time the host processor writes a word in the command_port of the interface, this interrupt is raised.
5. RST5.5: is used for single_step feature of the manager and basically provides a trap after every instruction.

Figure 15 depicts the I/O address map of the manager. Evidently, it currently supports only three types of I/O interfaces. Two of these Interfaces are implemented using two each of Parellel_Peripheral_Interface chips (Intel's 8255A-5). One of the Interface provides communication between the manager and the RTP by means of 3 input and 3 output undirectional ports. The other provides communication between the manager and the host_processor by means of 2 bidirectional ports, 2 unidirectional ports, and 2 control ports to support operation of the bidirectional ports. The only other I/O

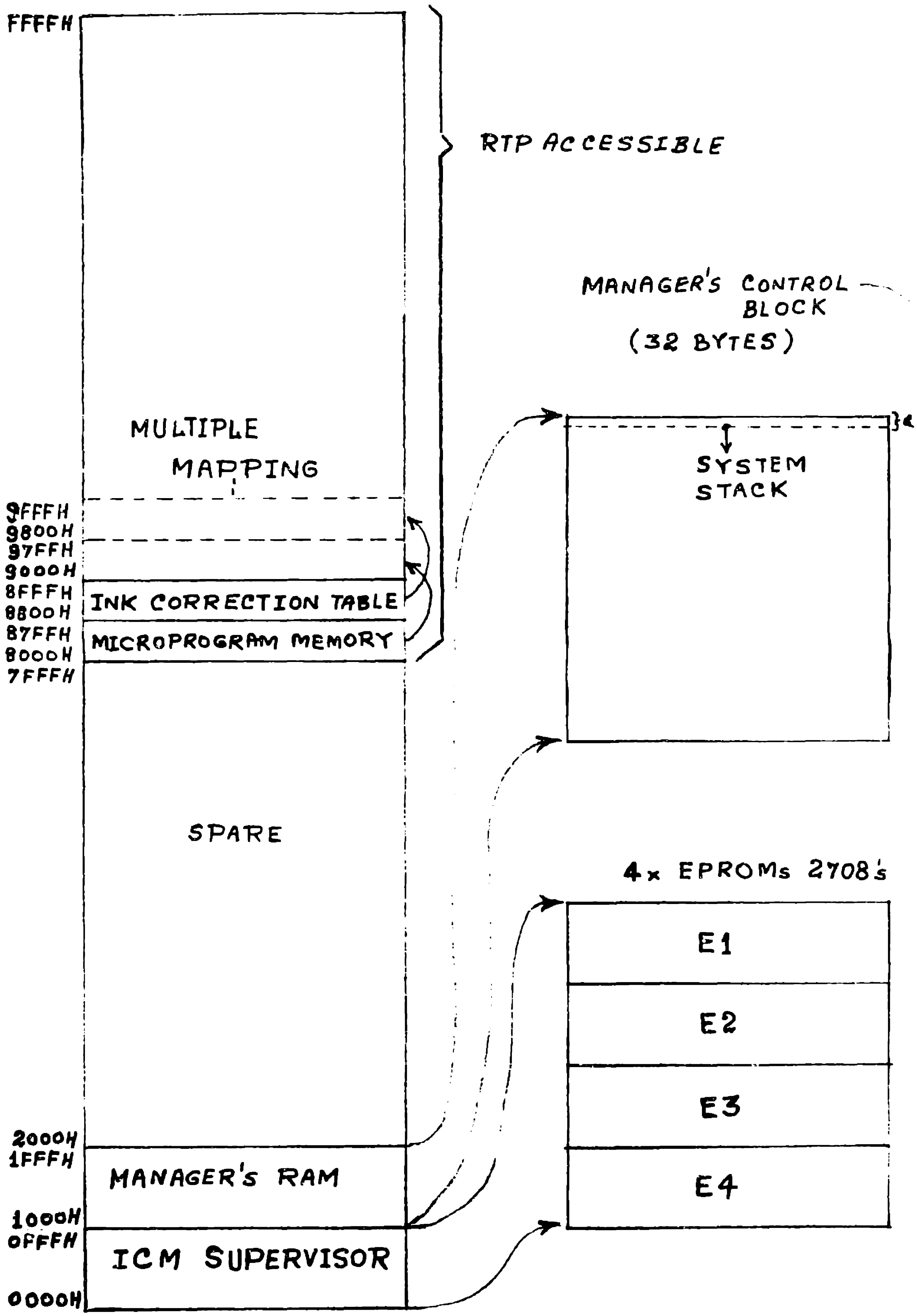


FIGURE 14 ICM MANAGER'S MEMORY ADDRESS MAP

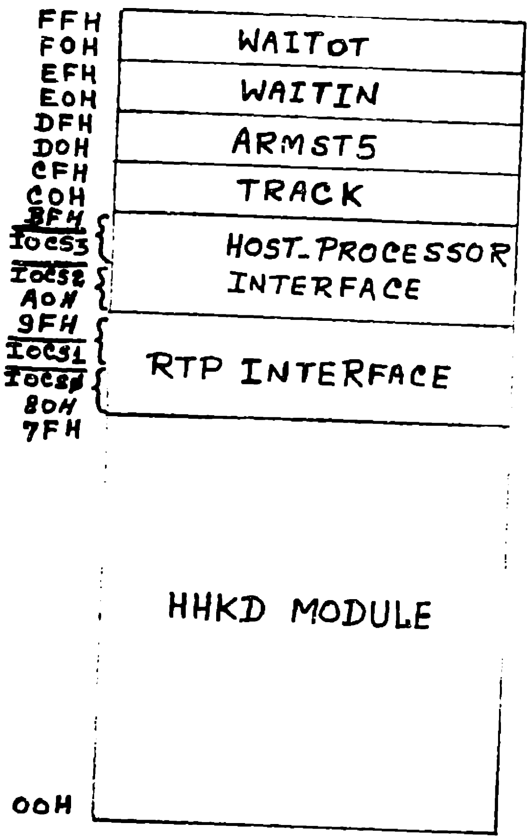


FIGURE 15 ICM MANAGER's I/O ADDRESS MAP

interface currently supported by the manager is the Programmable Keyboard/display controller chip (Intel's 8259), which is used to communicate with a hand_held keyboard/display module. This HHKD module, in combination with the operating software, provides the functionality of a low_cost development system such as SDK_85, in addition to providing many other functions. Since the operating environment of the RTP is completely under its software control, the manager lends itself to being a very powerful and flexible debugging/maintenance tool.

4.2. Software Architecture

Current research in engraving_phase color processing requires the Ink Correction Module to have the flexibility of a development tool. Therefore, the system must be architectured to provide as much functionality through software as possible. Designing the real_time processor as a microprogrammable machine and allowing a microprocessor (the manager) the ability to load/modify its control memory under software control, yields exactly the desired architecture. This approach essentially requires a smart software system to live in the manager. Some of the desirable characteristics of such a software system are as follows;

1. Must have stand alone capability (the ability to live by itself) in order to support essential functionality even in the absence of other modules.
2. Should provide power_up initialization.

3. Should provide access to an operator through a keyboard/display module.

4. Should be able to communicate with other software systems operating on the host_processor (PDP 11).

5. Should provide memory diagnostics

6. Should provide management and diagnostics of the Real_Time Processor.

7. Should provide a simple program development environment for 8085-code in the style of Intel's low cost development system SDK-85.

8. Should be well structured to provide easy future expansion.

The software system ICM Supervisor ICMS.8080 (suffixed 8080 to meet the requirements of UNIX cross_assembler MICAL, which is designed for 8080)
was created to meet the needs of ICM system manager. While ICMS.8080 provides extensive operator interaction through HHKD module (handheld keyboard and display module), the primary user interface is through a software system ICMON (Ink Correction Monitor) operating on the Host Processor. It was desirable to do so because a console terminal, attached to host_processor, is a centralised user interface in the existing system. Another important reason for this architecture is that the ICM data_bases would eventually live in the file system (secondary storage) of the host processor and a software package would anyway be required in the host system to download the program/data objects into ICM.

Accordingly, ICMON is designed as a command Interpreter, which, in communication with ICMS, performs the following function in response to commands typed on the console terminal;

1. Read from and write to Terminal Console.
2. Transfer arbitrary length records between the host_processor and the ICM manager in either direction.
3. Activate a variety of actions in the Real_time operation.
4. Read ICM status.
5. Fetch information on error condition.
6. Perform diagnostics on RTP.
7. Provide miscellaneous utility such as memory dump, load memory from console as well as paper tape etc.

The software development associated with ICM requires programming in MACRO-11 (PDP-11 assembly language), 8085-assembly language as well as special assembly language or machine language corresponding to the Real_time processor. To support these program development and debugging activities, a number of utilities must be provided on the system.

Following utility programs were created in addition to many already supported by CIPG's UNIX system.

1. Intel Hex Code Formatter
2. Hex_to_Binary Converter
3. Micro_assembler for RTP

In view of the difficulty of microprogramming a two_level pipeline_architectured machine, it was considered desirable to create a micro assembler for RTP so that microprogramming could be done in symbolic language. MICRASS, RTP's microassembler, is currently under development by another student.

CHAPTER 5

Hardware Organisation

Hardware of ICM is organised as a set of 9 wire_wrapped boards, 6 quads and 3 duals which plug into a DEC_style 9 slot system unit as depicted in figure 16. Slots A and B of the system unit back_plane are wired across as UNIBUS. ICM system manager is implemented using just one quad plus one dual board (bard#7A&#7B) whereas the rest of the boards constitute the Real_time Processor. Figure 17 depicts the signal information map of the ICM backplane. Appendix B presents a photographic view of the actual hardware. All the logic diagrams of ICM are filed under corresponding board numbers in the ICM documentation.

5.1. Real Time Processor

The RTP consists of 5 quad and 2 dual_boards as described below;

5.1.1. Block Diagram Description Figure 18 gives the block diagram, depicting the organisation of the Real_Time Processor. The processor constitutes of the following modules;

1. Data Inerface
2. Microprogram Controller
3. Microprogram Memory

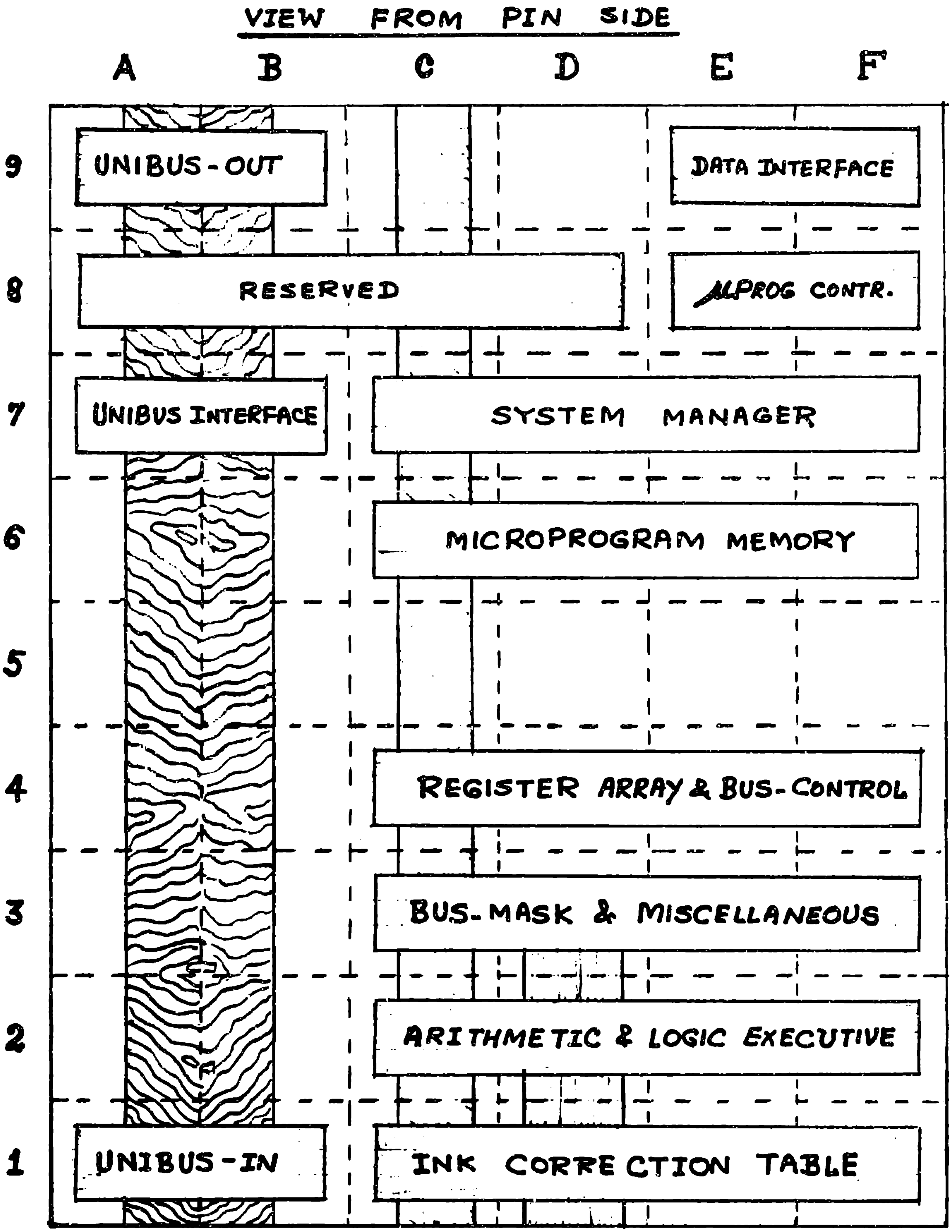
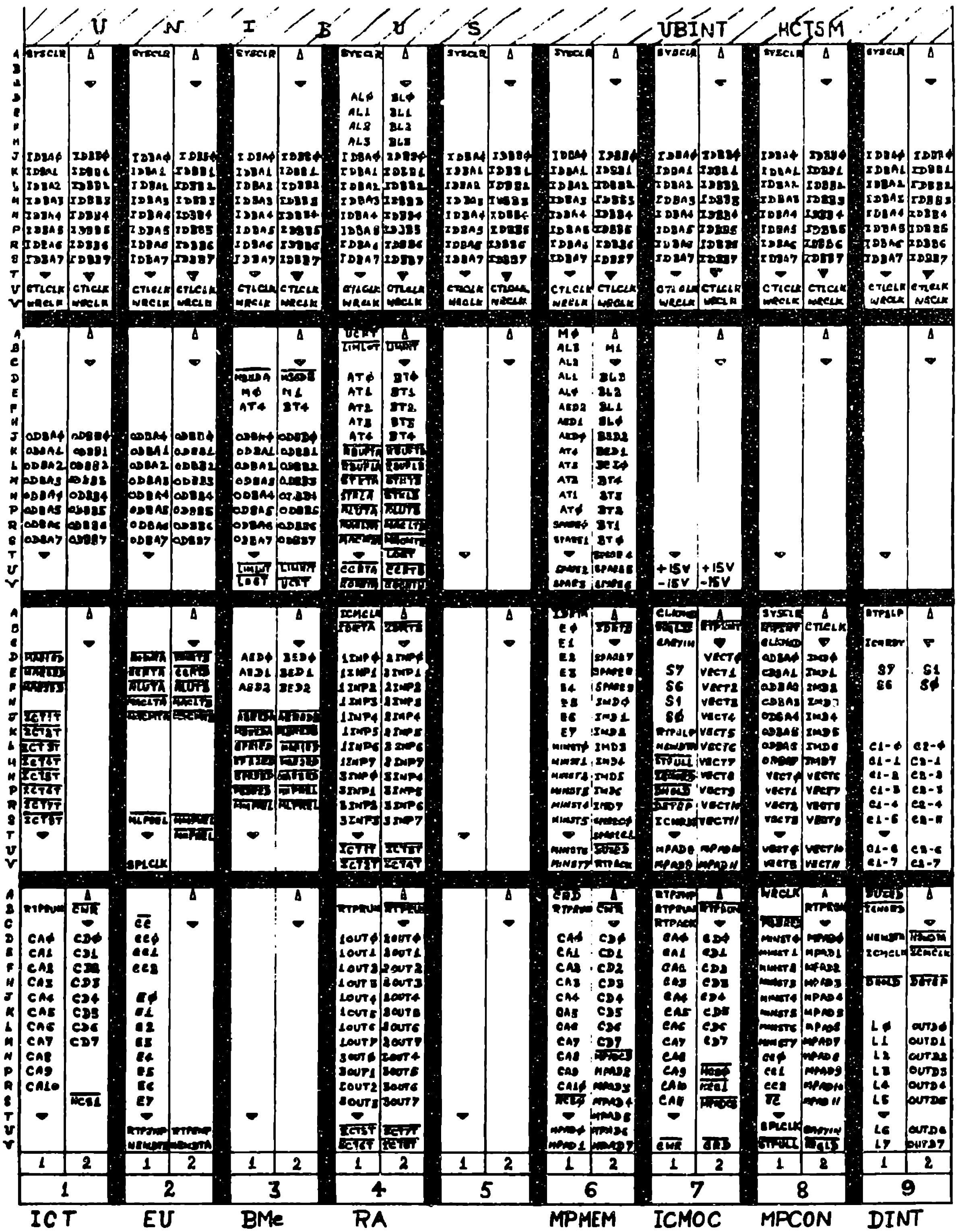


FIGURE 16 ICM SYSTEM UNIT LAYOUT

FIGURE 17 SIGNAL MAP OF ICM BACKPLANE



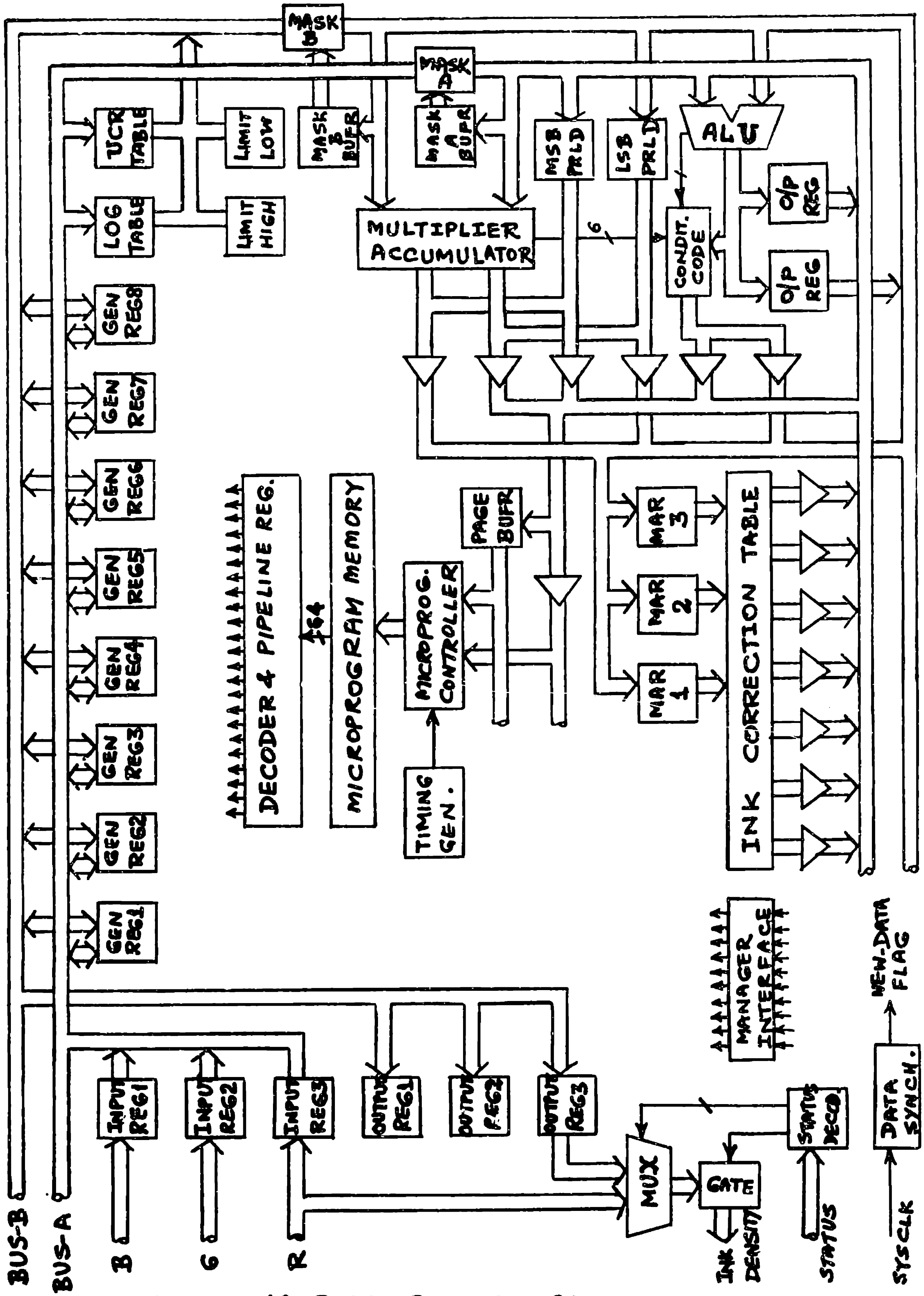


FIGURE 18 BLOCK DIAGRAM OF REAL-TIME PROCESSOR

4. Register Array & Bus Control

5. Bus Mask and Miscellaneous

6. Arithmetic and Logic Executive

7. Ink_Correction Table

The data interface module interfaces the real_time processor to the engraving system. RTP has 3 input and 3 output registers for communication with the outside world. There are eight genenral purpose registers which can both listen and talk to any of the two data_buses A and B. The buses themselves are segmented into two sections viz (i) Input section and (ii) output section. This facilitates buffering as well as allows hardware masking. The masks reside in registers and can be manipulated by the microcode dynamically. The input section of the buses are tied to I/O registers and the general registers, whereas, the output section is connected to the elements of ALE (Arithmetic and Logic Executive) and the ICT (Ink_Correction Table). The ALE has 3 processing elements viz (i) an 8 by 8_bit MAC (Multiplier Accumulator) (ii) A 8_bit ALU and (iii) an 8_bit/16_bit ROSH (Rotator_Shifter). The Ink_Correction Table has 3_address registers, memory and data buffers. ICT is also accessible by the manager.

The microprogram controller generates the next sequential address for the microinstruction stream as well as provides powerful branch and sub_routine call capabilities. Branch addresses may be selected from 3 external sources or 2 internal sources. Branches may be conditional or unconditional as explained in section 5.1.2.2 and 5.1.2.6. The microprogram controller is driven in a two_level pipelined architecture, clocked at 6.25 MHz. The RTP clocks, viz. WRCLK and CTLCLK are controlled by the system manager, which also has the capability to single_step the RTP. The microprogram memory is addressed by the microprogram controller via a pipeline registerand is extendable in depth. Like ICT, the microprogram memory is also accessible by the manager in off_line mode. The manager views these memory segments as byte organised and 2k deep. Whereas RTP's view of microprogram memory is 256 deep by 64_bit wide and that of ICT is 512 deep by 32_bit wide. The 32_bits data from ICT are, however, formatted as 7 words as explained in section 5.1.2.7.

<u>5.1.2.</u> <u>Apparatus Description</u> Following is a hardware description of the various modules which constitute the Real_time Processor.

<u>5.1.2.1.</u> <u>Data Interface</u> dual board (#9), implementing this module, carries one 50_pin and two 40_pin flat cable connections for interconnecting the RTP with the rest of the engraving_phase system. The cable connections are shown in figure 11.

All those signals, which do not have any interaction with ICM are directly connected between the input connector and the output connector. The "Green" and "Blue" channel data are connected to the Input registers 1 and 2 (on register array board) respectively via the backplane. The "Red" channel data is connected to Input Register 3 (on Register Array board) via the back plane as well as an on_board multiplexer . The function of multiplexer is to provide a bypass data path in the transparent mode. The status command from color Data Formatter is loaded in the status register with every ICM clock. The meaning of status bits is explained in figure 10. The inhibit_bits in the status_word is compared with the color_mode to make a decision whether the outputs should be inhibited for the current process_cycle. If the inhibit_bit for a color is set and the color mode corresponds to the same color, output is inhibited by pushing all outputs bits to "HIGH" corresponding to the "drop out" level.

ICM clock is generated by ORing the INITCLK with SYSCLK and defines the start of a process_cycle. The rising edge of ICM clock loads input data in the Input registers and also sets a flag signalling to the microprogram that new data has arrived. When RTP finishes the computation during a process_cycle, the microprogram keeps testing this flag bit in a loop so that when the flag is set a new computation cycle is initiated.

<u>5.1.2.2.</u> <u>Microprogram</u> <u>Controller</u> is implemented on a dual_board (#8). The function of the microprogram controller is to generate the address of the next micro_instruction. This functionality is produced by a versatile LSI chip viz. Am2910 in the following manner. Am 2910 takes 4 instruction bits to decide how the next address will be selected, a detailed description of which is presented in AMD Data Book [26]. The D_input to Am2910 has a selection from 3 possible

sources. MAP selects an address from the A_bus, therby making the entire architecture of RTP available for address manipulation. PL selects an address from the Immediate operand field of micro_instruction and thus provides immediate branch capability. VECTOR selects an address vector from the system manager and is useful for inrerrupts. By convention, location zero in the microprogram memory always contains a JUMP_VECTOR instruction, so that if a hardware interrupt from manager is caused, JUMP_ZERO instruction is jammed into AM2910, thus fetching a JUMP_VECTOR instruction. The next microcycle then causes an unconditioned branch to the vector address specified by the manager. The manager may specify a 12_bit vector address and thus cause a jump to any address in the entire 4k address space. Whereas, the PL and MAP use a 4_bit page address held in a page buffer register which gets concatenated to the 8 bits from Bus_A or the IDR. Thus, this mechanism provides a paged_memory view of the 4k microprogram address space offering many new possibilities. Page buffer may be written from bus_A. The output from Am 2910 is pipelined via a set of registers. The output of the pipeline register may be viewed as the micro_program counter since the micro_instruction out of the memory is also pipelined. (PC)+ and the MPCI (Micro Program Controller Instruction) are locally displayed for the sake of debugging and maintenance.

This module also generates the two RTP_wide clocks viz. WRCLK and CTCLK. Both clocks have exactly the same frequency of 6.25 MHz, derived from a crystal controlled 25 MHz base clock. CTLCLK is just a 20_ns delayed version of the WRCLK. The entire control structure of RTP (the instruction pipeline registers) is clocked on the rising edge of CTCLK. Whereas all data transfers occur on the rising edge of WRCLK. 20_ns delay of CTCLK thus allows ample hold_time for the data_holding devices. The clock drivers may be disabled and single_stepped by the manager by executing the following sequence;

1. SYSCLR:= LOW; enable data registers
2. RTPRUN:= LOW; kill WRCLK & CTCLK
3. CLKMOD:= HIGH; set "single_step" mode
4. RTPRUN:= HIGH; single step RTP
5. RTPRUN:= LOW ;
6. CLKMOD:= LOW; reset clock mode to "RUN"

<u>5.1.2.3.</u> <u>Micro</u> <u>Program</u> <u>Memory</u> is implemented on a quad board (#6). This memory is implemented using 16 high_speed bipolar memory chips 93422 from (fairchild). The organisation of each chip is 256x4. During the real_time mode the memory is addressed by the microprogram controller's pipeline register MPADO_7 and is not accessible by manager (RTPRUN being HIGH). 64 data bits are simultaneously available from this memory in this mode, which constitute a micro_instruction. The format of the micro_instruction is described in section 5.1.3. During off_line mode (RTPRUN = LOW), it is seen as a 2kbyte memory segment by the manager and may be read or written by the 8085 MPU just as its own local memory.

5.1.2.4. Register Array and Bus Control are implemented on a quad board (#4). This module contains eight general registers which are implemented using dual_port register chips DM 8542 N (National Semiconductors). Any of the register could be written from any of the two buses while, simultaneously, any register or some other element may be outputting some data onto these buses. Use of these devices eleminate any programming constraints on how these registers may be accessed during a transaction.

This module also contains the decoders (74154's) and the pipeline registers (74S175's) for bus_control. The decoding logic ensures that no two devices may talk on the same bus at the same time.

The Input/Output registers are implemented using 74LS173N. As is to be expected, Input registers are read_only, whereas, the output_registers are write_only by the microcode.

5.1.2.5. Bus Mask and Miscellaneous are implemented on a quad board (#3) . This board contains the following items;

1. Gates, buffers and registers to implement the programmable bus_masks.
2. Decoder and pipeline registers for controlling eavesdropping function.
3. A buffer which outputs `FF' (LIMITHI).
4. A buffer which outputs hex `00' (LIMITLO).
5. Non_linear operator LOGT.
6. Non_linear operator MUCR.

Masks (arbitrary bit patterns) may be loaded in mask buffers which are implemented with two 74LS379, for each bus. Depending upon the direction of buffers, data from one side of the bus is ANDED with the MASK and output to the other side of the bus. The eavesdropping function (which essentially means that data can be transferred to more than one device during a microcycle) is provided by two 3_bits field, one for each bus. Decoders and pipeline registers provide the control signals for the eavesdropping function. The two buffers `FF' and `00' provide the data for clamping results when overflows and underflows occur in an arithmetic operation. LOGT and MUCR look_up tables are implemented by 74S471 bipolar PROMS. Both the non_linear operators are addressed by the A_bus and output data to the B_Bus. This board also has room for future expansion.

5.1.2.6. Arithmetic and Logic Executive is implemented on a quad board (#2). ALE consists of following 3 elements;

1. Rotator_Shifter (ROSH)
2. Multiplier_Accumulator (MAC)
3. Arithmetic and Logic Unit (ALU)

ROSH is implemented using multiplexers and the data paths are structured in a manner such that it can be operated in either 8_bit `byte' mode or 16_bit `word' mode. In byte mode contents of bus_A and contents of bus_B are either shifted or rotated, left or right, as two 8_bit bytes and the result is left in ROSH's output register. In word mode, the contents of both buses are concatenated (Bus_B is more significant) and either shifted or rotated,left or right, as 16_bit word and the result is left in the same way. This special structure of ROSH allows MAC operation on real numbers by providing the ability to justify and align data properly.

The Multiplier_Accumulator is a high_speed bipolar LSI chip (TDC1008J from TRW). This element provides the capability to multiply two 8_bit numbers in parellel as well as allow accumulation of successive procucts. MAC has a substract mode also, in which the last cumulative product may be subtracted from the current prduct. The output of MAC is a 19 bit result. However, hardware is configured such that only 16_bit results can be output to Bus A and Bus_B. The higher_order 3_bits of MAC output are input to condition_code Register and can be tested for conditional branch. MAC can be set up to do either sign_magnitude or 2's complement arithmetic with results being truncated or rounded_off. WRCLK is used to latch the inputs operands as well as the control signals to MAC. A special clock is generated to clock the output register as shown in the time diagram presented in appendix C.5. In "accumulate" modes the computation through MAC is pipelined. Clocking of data in and out of MAC imposes certain constraints on the programmer, which are discussed in section 5.1.5.1. The output registers of MAC may be pre_loaded using pre_load registers. The program must load the pre_load registers first with appropriate data before executing a pre_load micro instruction.

The ALU is constructed with 74S181 ALU chips. Any mode and instruction may be used. However, the condition code logic for overflow bit is designed to be valid for only a subset of operations as described in section 5.1.5.3. The condition code register (CCR) is designed in a way such that the condition code resulting from previous arithmetic operation is saved, Only if the current instruction specified an ALU or a MAC operation. Any other instruction leaves the CCR undisturbed. This helps in multiple_way branches.

5.1.2.7. Ink Correction Table is implemented on a quad board (#1). ICT is very similar to microprogram memory in many ways, but has a different functionality. It is essentially a memory board implemented with 16 high_speed bipolar memory chips (93422 of Fairchild). The organisation of the chip is 256x4. During the real_time mode (RTPRUN = HIGH) the memory is addressed by a set of Memory Address Registers (MARs) and is not accessible by manager. 32 data bits are simultaneously available from this memory in this mode. These 32 bits provide the correction data in the form of 7 elements as shown in Table 1. All the seven elements are formatted as 8_bit bytes, properly justified, before being read on bus_A. MARs can be written from bus_B only. During off_line mode (RTPRUN=LOW), ICT is seen as a 2k_byte memory segment by the manager and may be written by the 8085_MPU.

BYTE REF.	MEM BIT REF.	PARAMETER	ICT REF.	BIT #	SCHEM REF.
3	31	T'	ICT1	6	m6
3	30	T'	ICT1	5	m5
3	29	T'	ICT1	4	m4
3	28	T'	ICT1	3	m3
3	27	T'	ICT1	2	m2
3	26	T'	ICT1	1	m1
3	25	T'	ICT1	0	m0
3	24	$\Delta\tilde{T}_T$	ICT2	5	n5
2	23	$\Delta\tilde{T}_T$	ICT2	4	n4
2	22	$\Delta\tilde{T}_T$	ICT2	3	n3
2	21	$\Delta\tilde{T}_T$	ICT2	2	n2
2	20	$\Delta\tilde{T}_T$	ICT2	1	n1
2	19	$\Delta\tilde{T}_T$	ICT2	0	n0
2	18	$\Delta\tilde{T}_{T+1}$	ICT3	4	p4
2	17	$\Delta\tilde{T}_{T+1}$	ICT3	3	p3
2	16	$\Delta\tilde{T}_{T+1}$	ICT3	2	p2
1	15	$\Delta\tilde{T}_{T+1}$	ICT3	1	p1
1	14	$\Delta\tilde{T}_{T+1}$	ICT3	0	p0
1	13	$\Delta\tilde{T}_{T+2}$	ICT4	4	q4
1	12	$\Delta\tilde{T}_{T+2}$	ICT4	3	q3
1	11	$\Delta\tilde{T}_{T+2}$	ICT4	2	q2
1	10	$\Delta\tilde{T}_{T+2}$	ICT4	1	q1
1	9	$\Delta\tilde{T}_{T+2}$	ICT4	0	q0
1	8	$\Delta\tilde{T}_{Y,C}$	ICT5	2	r2
0	7	$\Delta\tilde{T}_{Y,C}$	ICT5	1	r1
0	6	$\Delta\tilde{T}_{Y,C}$	ICT5	0	r0
0	5	$\Delta\tilde{T}_{C,M}$	ICT6	2	s2
0	4	$\Delta\tilde{T}_{C,M}$	ICT6	1	s1
0	3	$\Delta\tilde{T}_{C,M}$	ICT6	0	s0
0	2	$\Delta\tilde{T}_{M,Y}$	ICT7	2	t2
0	1	$\Delta\tilde{T}_{M,Y}$	ICT7	1	t1
0	0	$\Delta\tilde{T}_{M,Y}$	ICT7	0	t0

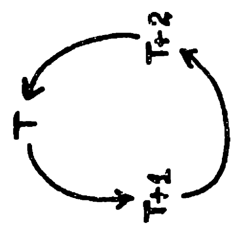
<u>Example</u>

(1) If T=Y Then T+1=C, T+2=M

(2) If T=C Then T+1=M, T+2=Y

(3) If T=M Then T+1=Y, T+2=C

CONVENTION:



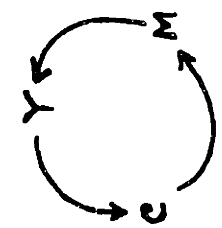


TABLE 1. DISTRIBUTION OF DATA BITS IN INK CORRECTION TABLE

5.1.3. MICRO instruction Format RTP's microinstruction is formatted as a horizontal_vertical combination. While designing the intention was to format horizontally as much as possible in order to support concurrency. However, mutually exclusive operations were grouped together and vertically coded. Thus, 64_bit micro_instruction word is divided into 8 functional groups as follows;

1. Bus_A control — 12_bits
   - who talks - 5 bits
   - who listens - 4 bits
   - who eavesdrops - 3 bits
2. Bus_B control — 12_bits
   - who talks - 5 bits
   - who listens - 4 bits
   - who eavesdrops - 3 bits
3. Microprog Controller Instr. — 8 _ bits
   - Instruction to Am 2910 - 4 bits
   - CCEN to Am 2910 - 1 bit
   - CCMUX select - 3 bits
4. Instruction to ALE — 8 - bits
5. Immediate operand — 8 - bits
6. Mask Enable — 2 - bits
7. Spare — 10 - bits
8. Special control — 4 - bits

Appendix D presents in detail the assignment of memory bits and how the vertically formatted fields must be decoded to develop all the control signals.

5.1.4. Functional Description All data transfers, between the various elements, take place on either Bus_A or Bus_B which constitute the dual data_bus structure. Every device, connected to these buses, has 3_state outputs, which remain in high_impedance state until commanded by the control structure to source data on the buses. All data transfers on the buses occur synchronously with RTP's system clocks, viz WRCLK (Write_Clock) and CTLCLK (Control_Clock). CTLCLK has exactly the same frequency as WRCLK (6.25 MHz derived from a crystal oscillator of 25 MHz) but has a slight phase delay to meet the worst case hold_time requirement for every device in the system. The entire control structure of RTP is clocked on the positive transition of the CTCLK whereas all the devices sink data on the positive transition of the WRCLK. Elements, such as general registers, which sink data selectively, are implemented with devices which are gated for data sink. However, many devices in the system (such as arithmetic processor's output register's) are not gated and hence listen during every microcycle. This implies that these devices will contain garbage during other times (when not being used) of program flow. However, this does not cause any problem because the program sequence must pick the result from these devices at the appropriate time with reference to the input. Appendix C presents a set of time diagram illustrating various bus_transactions. Essentially, a bus transaction is caused by switching of control signals such that one of the device on the bus sources the data, while the destination device sinks it.

5.1.5. Hardware Constraints on Programming Owing to certain hardware limitations and/or design trade_offs, certain constraints are imposed on the programmer as described below;

5.1.5.1. Mac Transaction Since MAC is a clocked device, its timing must be derived in such a manner that its operation ties up appropriately with the rest of the processor timing as well as maximum thruput is obtained. Since the inputs must be clocked in and at least 100 nanoseconds propagation time must be allowed before the result can be clocked into the output register, MAC operations can not be timed like other processing elements and require more than one microcycle to do even a single multiply. However, for successive MAC operations, such as accumulate, processing may be pipelined through MAC to yield higher thruput. Thus, whereas 2 microcycles are necessary for a multiply operation, 3 microcycles would suffice for a multiply operation followed by multiply_accumulate operation. The timing for various MAC transactions is illustrated in appendix C.5. Therefore, from programmer's point of view, the only constraint imposed is that every single or successive sequence of MAC operation must be followed by a NOP instruction, before the result is obtained.

5.1.5.2. Branch Instruction As mentioned earlier, RTP has a 2_level pipelined microprogram control architecture. This implies that no matter what instruction the microprogram controller may currently be executing, the next micro_instruction, being already in the pipeline, will get to it for execution. This is perfectly all right when instructions are being executed successively. But if a branch instruction is executed, an undesirable side effect of pipelining occurs. Therefore, to maintain proper program flow, it is necessary that each branch instruction, conditional or unconditional must be followed by a NOP. This overhead, which is natural in a pipelined architecture, is acceptable in the interest of higher thruput resulting from pipelining.

5.1.5.3. Conditional Branch on Overflow For the sake of simplicity, ALU's condition code logic was designed in a manner such that the overflow bit is valid only for the following arithmetic operations;

1. A plus B
2. A minus B
3. A plus 1
4. A minus 1
5. A plus A
6. A plus A plus 1

The programmer must recognize this fact if a conditional branch on overflow is used.

5.1.5.4. Non_linear Operation Non_linear operations viz. LOGT and MUCR are mapped through look_up tables which are implemented by 74S471 PROMs. Even though these are high_speed schottky PROMs, these have a longer worst_case access time than most other elements in the RTP. A design trade_off was made to keep the microcycle time small and allow two microcycle for these non_linear operations, because such operations are not used very frequently anyway. Therefore, the programmer must code two successive instructions for such operations.

5.1.5.5. Hardware Masking The Bus_Mask hardware has already been described in section. It is necessary that a mask be programmed by loading appropriate bit pattern in the mask buffer, before any masking operation is done. The masks would remain undisturbed unless overwritten by another bit pattern.

5.1.5.6. ICT Look up In real_time mode, ICT is addressed by a set of three Memory Address Registers, each 3_bit wide. To obtain any useful output from the Ink_Correction Table, the MAR's must be set up atleast one microcycle ahead.

5.1.5.7. Input/Output The I/O operations in RTP are always through the Input and Output Registers. However, the manager may pass some data arguments through the instruction stream in the form of immediate operands, although, such indirect data transfers appear to be useful for diagnostic purposes only.

5.1.5.8. Interruption The real_time Processor can be interrupted only by the manager. The interrupt is caused by jamming a JUMP_ZERO instruction to the Microprogram Controller. By convention, location zero in memory must contain a JUMP_VECTOR instruction. The interrupt vector is supplied by the manager and is programmable. The manager is thus able to steer the program flow to any arbitrary segment by causing an interrupt. This feature, however, must be used very carefully because currently the RTP does not stack the PC on interrupt and thus a return to the executing program is not automatic. In other words, on an interrupt, RTP aborts the current task and switches to a new task.

5.2. ICM Manager

The manager is a 8085_MPU based microcomputer. Figure 19 presents a block diagram of the system manager.

5.2.1. Block Diagram Description The system manager has 4K of program memory (PROM), 4K of data memory (RAM) and supports three types of interfaces as mentioned below;

1. RTP Interface.
2. UNIBUS Interface.
3. HHKD_Module Interface.

The rest of the manager's hardware consists of bus_buffers, address latch, chip select decoding logic and devices for implementing the interrupt circuits. +12V and -5V power supplies required for PROMs (Intel's 2708s) are derived from +15 and -15V Bus power supplies by on_board fixed regulators. Manager may be reset from one of three points. (i) on_board reset switch, which also has a power_up reset circuit. (ii) Unibus reset line, and (iii) a reset switch

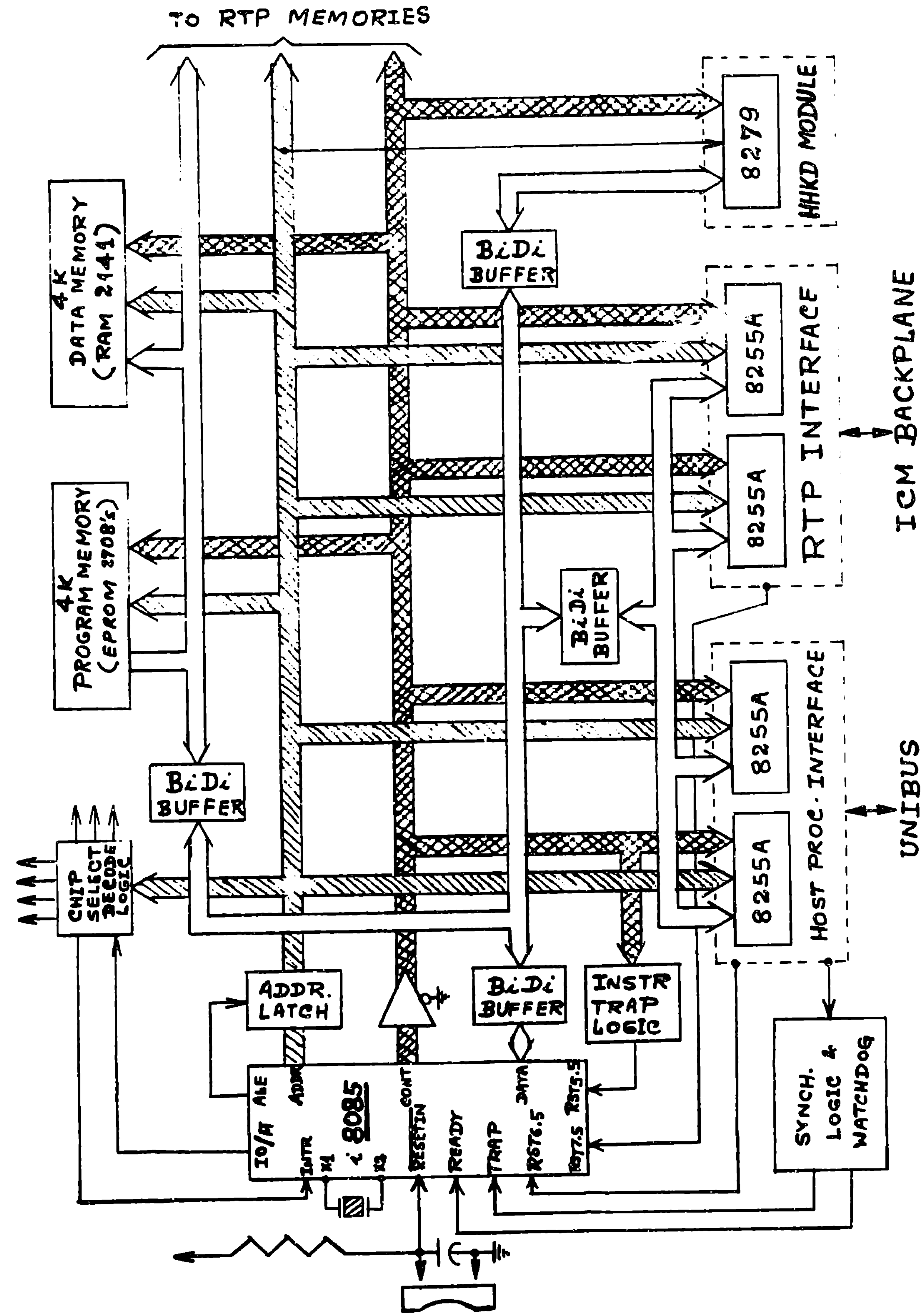


FIGURE 19 BLOCK DIAGRAM OF ICM SYSTEM MANAGER

installed in HHKD module.

The Interrupt and restart lines are used for the following function;

1. INTR: Trap for non_existent memory addressing
2. RST5.5: Single step facility of KHDMON.
3. RST6.5: Handshaking with ICMON
4. RST7.5: Monitor ICM status changes
5. TRAP: Whtch dog for UNIBUS hanging_up

5.2.2 Apparatus Description

5.2.2.1. MPU, Memory and RTP interface are implimented on a quad board. The manager's data_bus is partitioned between memory and I/O spaces. Two sets of bidirectional data_bus fuffers, enabled either by IO/$\overline{M}$ signal or its complement, partition the address space into memory and I/O spaces respectively. All read and write timings of manager's MPU is standard as per MCS_85 user's manual. All signals out of the MPU are properly buffered. The top 32_K byte memory space is used for RTP's microprogram memory and the Ink_Correction Table, even though, only 4K bytes of memory is used. The entire 32_Kbyte address space is used to simplify chip select decoding logic. A control line called RTPRUN, from the RTP Interface, must be set `LOW' for manager's MPU to read or write RTP's memories.

The manager's I/O space contains four 8255s and one 8279 (off_board) in addition to a few addresses being used to generate some special function control signals as follows;

1. TRACK(L) - Resets watchdog on Trap input.
2. ARMST$(L)- Resets Instruction trap circuit.

3. WAITIN(L)- Synchronizes input transaction to manager.

4. WAITOT(L)- Synchronizes output transaction from manager.

Two 8255's are used in mode `0' for implementing RTP Interface as follows;

1. Port A1 (C4) - unidirectional
2. Port B1 (C4) - unidirectional
3. Port C1 (C4) - unidirectional
4. Port A2 (C2) - unidirectional
5. Port B2 (C2) - unidirectional
6. Port C2 (C2) - unidirectional

Table 2 gives an exhaustive listing of each signal and the corresponding interface reference. ICM backplane signal_map, figure 13, shows how these signals are connected to RTP.

The HHKD_module Interface is implemented through a programmable keyboard_display controller which is located off_board (described in section 5.2.1.3). All signals to 8279 are buffered on_board in a manner such that if the switch K1 is off, then all lines are tristated and +5V power supply is cut_off. When K1 is switched `on', +5v supply to the HHKD module is connected and also the tristate gates are enabled. Also, enable control is interlocked so that if the HHKD module is not attached the signals remain tristated This elaborate arrangement is provided to protect 8279 when HHKD module is attached without switching off the power supply.

RTP PORT 1			
PORT REF.		SIGNAL NAME	HARDWARE REF.
DA1RTP	PA0	VECT0	C4 - 4
	PA1	VECT1	C4 - 3
	PA2	VECT2	C4 - 2
	PA3	VECT3	C4 - 1
	PA4	VECT4	C4 - 40
	PA5	VECT5	C4 - 39
	PA6	VECT6	C4 - 38
	PA7	VECT7	C4 - 37
DB1RTP	PB0	S0	C4 - 18
	PB1	S1	C4 - 19
	PB2	S2	C4 - 20
	PB3	S3	C4 - 21
	PB4	S4	C4 - 22
	PB5	S5	C4 - 23
	PB6	S6	C4 - 24
	PB7	S7	C4 - 25
DC1RTP	PC0	RTPRUN	C4 - 14
	PC1	ERROR	C4 - 15
	PC2	SYSCLR	C4 - 16
	PC3	RTPINT	C4 - 17
	PC4	CLKMOD	C4 - 13
	PC5	CARRYIN	C4 - 12
	PC6	REGLOAD	C4 - 11
	PC7	RTP JMP	C4 - 10

RTP PORT 2			
PORT REF.		SIGNAL NAME	HARDWARE REF.
DA2RTP	PA0	IDBA0	C2 - 4
	PA1	IDBA1	C2 - 3
	PA2	IDBA2	C2 - 2
	PA3	IDBA3	C2 - 1
	PA4	IDBA4	C2 - 40
	PA5	IDBA5	C2 - 39
	PA6	IDBA6	C2 - 38
	PA7	IDBA7	C2 - 37
DB2RTP	PB0	IDBB0	C2 - 18
	PB1	IDBB1	C2 - 19
	PB2	IDBB2	C2 - 20
	PB3	IDBB3	C2 - 21
	PB4	IDBB4	C2 - 22
	PB5	IDBB5	C2 - 23
	PB6	IDBB6	C2 - 24
	PB7	IDBB7	C2 - 25
DC2RTP	PC0	VECT8	C2 - 14
	PC1	VECT9	C2 - 15
	PC2	VECT10	C2 - 16
	PC3	VECT11	C2 -17
	PC4	IGNORDTA	C2 - 13
	PC5	DATAHOLD	C2 - 12
	PC6	DATASTEP	C2 - 11
	PC7	ICMRDY	C2 - 10

TABLE 2. RTP INTERFACE SIGNAL LIST

5.2.2.2. Host Processor Interface is implemented on a dual- board. This interface is also called the unibus Interface because the host processor is always intended to be a PDP_11 computer. The ability to communicate with the unibus is provided by using two 8255's each configured in mode `2' and mode `0' as follows.

1. Port A1 (S5) - Bidirectional
2. Port B1 (S5) - Unidirectional input
3. Port C1 (S5) - Control port for A1
4. Port A2 (S3) - Bidirectional
5. Port B2 (S3) - Unidirectional input
6. Port C2 (S3) - Control port for A2

Configured in this manner, the two 8255's provide one bidirectional 16_bit port and one unidirectional 16_bit port. The biderectional port is used for parallel data communication while the unidirectional port serves as a command port. Port A1 and A2 are connected to Unibus through a pair of bidirectional buffers so that the direction can be switched without any conflicts. Both ends of the interface are protected from being hung_up by the other end in the following manner; If the ICM hangs up unibus, it recovers through its own bus_time out trap mechanism, with the software generating appropriate warning messages. If Unibus hangs up ICM, a watch dog circuit provides recovery, as described in section 6.2.3.5. This interface board is physically connected to the MPU's buses through a 26_core flat cable. Table 3 gives the pin assignment for this interconnection.

SIGNAL NAME	CONNECTOR 3M	CONNECTOR 3M	SIGNAL NAME
CD1	2	1	CD0
CD3	4	3	CD2
CD5	6	5	CD4
CD7	8	7	CD6
CRD(L)	10	9	GROUND
CWR(L)	12	11	RESET
CA0	14	13	GROUND
IOCS2(L)	16	15	CA1
GROUND	18	17	IOCS3(L)
OBFA(L)	20	19	IBFA
RST7.5	22	21	GROUND
GROUND	24	23	RESETIN(L)
GROUND	26	25	CCLK

TABLE 3. CONNECTOR J1 - PIN ASSIGNMENT

5.2.1.3. Hand held Keyboard Display Module This module is implemented in a modified calculator casing. A TI_30 calculator conveniently provided the keypad and the 8_digit 7_segment LED display and a handy casing for the implementation of this module. Photographic view of this module is presented in appendix B. The original electronics of the calculator was completely removed and a small wire_wrap board containing 8279 and logic gates as shown in figure 20, was installed in the space provided for calculator's battery compartment. A 40-pin Flat cable header was installed at the top edge of the casing and was wired to the small board installed inside. A reset push_button switch was installed in the hole where normally the plug of a power supply adapter would fit. The displays were wired through a set of current limiting resistors as shown in the logic diagram. 8279 provided almost all the logic necessary for this interface. This approach yielded a compact self contained keyboard and display module, which is portable enough to be carried in a hand and attached, through a flat cable, to the system bus of a 8085_MPU based microcomputer. Table 4 gives of pin assignments in respect of the interconnection between the system manager and the HHKD module.

Figure 21 shows how the keyboard of HHKD module is organised. The 40 keys are partitioned into three groups:

1. Keyboard/Display monitor
2. RTP operation
3. RTP Diagnostics

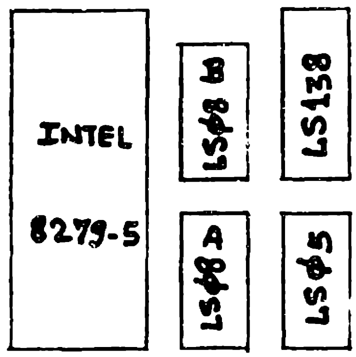


FIGURE 20 CHIP LAYOUT OF HHKD MODULE

0	8/H	MEM	ERST	DIGϕ
1/P	9/L	REG	RTST	DIG1
2/T	A	ADDR	RCLR	DIG2
3	B	STEP	VECT	DIG3
4	C	RUN	RRUN	DIG4
5	D	NEXT	RBRK	DIG5
6	E	BKPT	RSIN	DIG6
7	F	CLR	RHLD	DIG7

FIGURE 21 KEY LAYOUT OF HHKD MODULE

SIGNAL NAME	CONNECTOR 3M	CONNECTOR 3M	SIGNAL NAME
VCC	2	1	VCC
VCC	4	3	VCC
	6	5	
GROUND	8	7	CCLK
GROUND	10	9	RESET
GROUND	12	11	CRD(L)
GROUND	14	13	CWR(L)
GROUND	16	15	C/D
GROUND	18	17	IOCSL(L)
GROUND	20	19	RESETIN(L)
CD1	22	21	CD0
CD3	24	23	CD2
CD5	26	25	CD4
CD7	28	27	CD6
	30	29	
	32	31	
	34	33	
	36	35	
	38	37	
GROUND RET.	40	39	GROUND RET.

TABLE 4. CONNECTOR J2 - PIN ASSIGNMENT

The keyboard/Display monitor group contains all the hex character keys as well as the following command keys (concerned with the manager's operation only);

1. MEM - Memory
2. REG - Register
3. ADDR- Address
4. STEP- Single_Step
5. RUN
6. NEXT
7. BKPT - Breakpoint
8. CLR - Error_Reset

These command keys activate KHDMON section of ICM Supervisor and provide the functionality of a low cost 8085_based microcomputer development system such as SDK_85. Additionally, the RTST key provides RAM diagnostics.

The RTP management section contains following command keys;

1. RCLR - RTP System Clear
2. VECT - RTP VECTOR
3. RRUN - RTP Run
4. RBRK - RTP Breakpoint RUN
5. RSIN - RTP Single Step
6. RHLD - RTP Hold

The third section contains 8 keys for performing diagnostics which are explained in detail in chapter 8.

5.2.3. Functional Description The ICM manager functions much as a conventional microcomputer. It performs all the services through the algorithms specified by software. The manager should be operated by the software system called the ICM Supervisor. The manager is initialized by the power_up or by hitting the reset switch. During initialization all the 8255 modes for various interfaces are set as well as a number of house_keeping functions are performed as described in section 6.2.1. The manager may be called upon to perform a service function by invoking a command through the host processor or by hitting a command key on the HHKD module. In this case a prewritten routine from the program memory is executed by the manager. Alternatively, a user may write a program of his own, load it in the RAM area and execute it. However, if the latter approach is taken, great care must be exercised since damage to hardware may be caused if the 8255_port configuration is meddled with.

The hardware function of the interrupt circuits is discribed in section 6.2.3, so that the description of interrupt handling software becomes more understandable.

5.2.4. Programming and Hardware Constraints Since the system is configured completely under software control, some precautions have to be taken so that no damage may occur and useful results are obtained.

5.2.4.1. 8255 Port Configuration The 8255_ports are configured by the ICM Supervisor during the initialization sequence. These configurations are fixed and MUST NEVER BE CHANGED. The hardware around 8255's expects that the ports are configured as specified. Hence, if these port configurations are changed by writing any other mode setting in the control port, HARDWARE DAMAGE MAY RESULT. Therefore, any user who wishes to write his own code for the manager, MUST NEVER REFERENCE I/O ADDRESSES CHOST (A3), C2HOST (B3), C1RTP (83), C2RTP (193), C0279 (101). Sometime it may be desirable to twiddle individual bits of Port C in the RTP Interface. This can only be done by writing different bit patterns to either CIRTP or C2RTP. If one wishes to do this, one must carefully choose the bit patterns so that the modes are not altered.

5.2.4.2. RTP Memory Accessing As mentioned earlier, RTP memory may be accessed by the manager only during the off_line mode (RTPRUN=LOW). If one attempts to write RTP memory in real_time mode (RTPRUN_HIGH) then ERROR 4 condition would be set, signalling unsuccessful memory write operation. The original contents of the memory will, of course, remain unchanged.

If the manager attempts to read the RTP memory during the real_time mode, it would obtain garbage and no error condition is signalled. Hence, to read or write the RTP memories, the appropriate sequence is to halt the RTP first by setting RTPRUN=LOW and proceed thereafter.

5.2.4.3. HHKD Module Accessing HHKD module interface is designed in a way such that it should be possible to attach this module while the rest of the system is operating i.e. the system may not be powered down just for attaching this module. Since the +5V power supply for HHKD module is derived through the same cable, it is possible that the module's on_board chips (8279, in particular, being a mos device) may be damaged during this physical connection. For this reason, a switch (K1) is provided on the ICM manager board, which disconnects the +5V supply and tristates all input signals to HHKD module. An LED located next to this switch (K1) indicates whether the switch is `off' or `on' (bright LED indicates switch is `on'). Therefore, to protect HHKD module, switch should always be put `off' before attaching its cable to the ICM manager board.

CHAPTER 6

Software Organisation

The software for ICM is hierarchically organised at 3 levels. At the top level, the software system is resident in the host_processor (PDP_11) and is called ICMON (Ink_Correction Monitor), intended to provide the primary user_interface through a terminal. The next level in the hierarchy is an extensive microcomputer software package called ICMS.8080 (Ink_Correction_Module Supervisor) and its function is to operate the ICM manager. The third level constitutes the actual microprograms which may be downloaded either locally from manager's environment or from the host processor's environment (File System). In addition to these, system software is required to support program development at all the three levels. Some of these support software described in section 6.3 modules are.

6.1. ICM Monitor

ICMON is basically a command Interpreter designed as a conversational monitor. Appendix E presents a source listing of ICMON. All commands are processed by a library of routines. ICMON may be viewed as consisting of four major components as described below;

6.1.1. Command Process Library

ICMON is a highly structured program. It processes commands listed in its command table. When invoked, the ICMON reads a command word (upto 4 characters length) from the terminal, goes to process it and then returns to fetch the next command. It continues in this loop (of fetching a command and processing it) endlessly until an EXIT command is given, where upon, it returns the control to the parent operating system (PDP_11/10 version just HALTS, since it is designed to operate on a bare machine). A command word is fetched by GETCMD routine. Then, GOCMD routine makes a linear search through a command table to find a match. If a match is found the corresponding command process routine is called to do the job. The command process routine eventually return the control to GETCMD for next iteration. If no match is found, an appropriate warning is printed on the terminal.

Command process routines specify the exact action to be taken (described in the section 6.2.2). If any arguments are required to be specified by the operator, ICMON asks for it by name. Once all arguments are specified, the command is processed. After the process is done, the program prints the prompt message again, for the next command to be processed. `HELP' command generates a listing of all currently implemented commands.

6.1.2. Sub Routine Library Command Process routines require a number of primitive functions such as READ (from terminal), WRITE (to terminal), BN2OC (convert binary number to an octal string) etc. All such primitive functions are performed by sub_routines, which are supported as a library.

READ & WRITE sub_routines involve I/O operations. I/O on CIPG's PDP11/10 is not interrupt_driven. Therefore, in the current version, devices are polled to find out when I/O should be done.

6.1.3. Tables and messages ICMON maintains a table of index for all currently implemented commands. Future expansion of ICMON is extremely easy. To add a new command, a new entry may simply be added to table and the corresponding routine should be placed somewhere in the body of ICMON. A number of system messages are generated. These are also structured in a manner so that new messages can be added easily.

6.1.4. Exception Handling In the context of ICMON, only two types of exceptions may occur during operation viz. (i) stack overflow (ii) Bus_time out. When an exception occurs the handler routine generates appropriate warning as well as provides recovery. At first the cause of exception is determined. If it was due to stack overflow, the processor halts after giving a message. If not, the trap handler routine concludes that a bus_time-out must have occurred and provides a recovery from an otherwise hang_up situation, after printing apropriate warning messages.

6.2. ICM Supervisor

ICMS lives in the EPROMS (2708's) installed on the manager board. Presently 4 Kbytes (4 chips) of program memory are sufficient to accomodate the code generated from more than 2000 lines (excluding comments) of source code, a complete listing of which is presented in appendix F. To facilitate easy maintenance complete edit history is maintained as per the following convention; For any expansions added, only the subversion number may be changed. Thus, version 1.1 will be upward compatible with version 1.0 A change in version number denotes a more fundamental change which may destroy compatibility, meaning, version 2.0 may not provide all the functionality of version 1.19. All changes or expansions made in the software are always entered in the edit history.

ICMS broadly consists of 5 major sections, as described below;

6.2.1. Initialization Sequence activated by asserting RESET (L) signal either by the automatically by RC_circuit on power up, or manually by the reset push button. The initialization routine carries out the following tasks;

1. Resets watchdog on TRAP input.
2. Sets 8255 modes of RTP Interface.
3. Initializes RTP Interface.
4. Sets 8255_modes of Host_Processor Interface.

5. Sets 8279 mode in HHKD module.

6. Sets 8279 programmable clock.

7. Initializes system stack.

8. Initializes control block.

ICMS makes use of a 32_byte deep control block, located at the top of manager's RAM space (/1000 to /1FFF), to save all CPU's registers, program counter, stack pointer as well as system parameters. Parameters RAMPTR, BRKA, BRKD, DSPLM and RUNM are associated with the functioning of KHDMON section. Whereas, RPVECT, RPBRKD, RPSTRT and STATUS relate to the operation of real_time processor.

ERCODE contains a non_zero value in case of any error, which is used as a key to generate the message.

After initialization is done, the control gets transferred to KHDMON and stays with it until an interrupt occurs. When an interrupt occurs the control is transferred to the handler routine, which eventually returns it to KHDMON, after the interrupt has been serviced.

6.2.2. KHDMON (keyboard/Hex_Display Monitor) provides communication with ICM manager through the HHKD module in the following manner; It reads the keyboard for a command, processes it, displays the results and then waits for the next key. In this manner KHDMON is able to translate a set of commands from the keyboard into a series of actions. The keyboard and display are interfaced with the controller chip 8279. The program keeps testing 8279 flags in a loop to find out if there is anything in its buffer. As soon as a key is depressed, the numerical value of the key gets loaded in the 8279's FIFO and the buffer_not_empty flag is set. The CPU then reads the FIFO till it is empty. Once a key is read from 8279, the program determines which key was depressed. If it was a command key, then the corresponding routine is activated. The hex_keys enable arbitrary data objects to be defined and used as arguments for the command process routines. Some of the keys serve as double function keys. The functionality of these double_function keys depends upon the context defined by the keying sequence.

If a command key is depressed in the middle of another process, while a hex_key is awaited, the current command is aborted and the processing of new command begins. Upon entry, KHDMON first initializes the system stack and then initializes the display according to the flag stored in DSPLM (Display Node). Control is then transferred to GETCOM, which calls the KEY sub routine. The KEY sub routine reads the 8279 buffer in a loop until a key is depressed. If the buffer is not empty the KEY sub routine returns the value of the key, which was depressed. GETCOM compares this value with a known boundary to determine, if the key was a command key. If not, error condition `2" (signalling improper key) is set and a new call to KEY is made. GETCOM thus implements a loop, which filters out a command key. When a command key is detected by GETCOM, the offset into the command table is computed from the key-value and then address of the corresponding process routine is fetched. GETCOM saves the user's program counter in register_pair HL before the control is transferred to the command process routine. The capability provided by KHDMON, through the various command keys, is as follows;

CADDR Routine (Activated by ADDR Key);

This command provides the ability to point to any memory location and display its contents. When the ADDR key is depressed, the default value of user PC and its contents are initially displayed. However, the address may be modified by keying hex_characters. The address scrolls left, every time a hex character is keyed, the new key value appears (left most character is lost). Thus a new address is specified. The program automatically displays the contents of the memory location being pointed to by the address field at any time. The sequence of address modification is terminated by a new command key. The ADDR command chains with the NEXT command to display successive memory locations in a very convenient manner. Since, ADDR command does not have any side effects, many key sequences begin with this key.

CREG Routine (Activated by REG key);

This command displays the MPU's registers in the following manner. If the preceding key was a double- function hex_key (which should have succeeded the ADDR key in order to be recognized), the specified register name is displayed along with its contents. The display format is as follows; the first four digits display the user PC. The next digit displays the specified register name and the digit following it is always a 'dash'. The last two digits display the contents of the specified register.

This command also allows the contents of any register to be modified by keying in hex characters. The contents change in a scroll mode as the hex keys are depressed. If only one hex_character is keyed, the significant digit of the byte is taken to be zero by default. The sequence of contents modification is terminated by a new command key. As in the case of ADDR command, REG command also chains with NEXT command to display all other registers successively in a scrolled manner.

CMEM Routine (Activated by MEM key);

This command provides two types of function as determined by the number of hex_keys preceding this key. If a single double_function key preceded (which should have succeeded the ADDR key in order to be recognized, this key, the register_pair specified by the double_function key is displayed. The display format is as follows; the contents of the register_pair (a pointer) are displayed in the first four digits. The next two digits display the name of the specified register pair and the last two digits display the contents of the memory location pointed to by the contents of the register_pair (the pointer).

This command allows modification of the pointed memory location in much the same manner as that done by the REG command, as well as chains with NEXT command to display all other register pairs successively in a scrolled manner. The register pairs which can be displayed in this mode are namely, the Stack Pointer, Stack Top, Register_pair HL, Register_pair BC and Register_pair DE.

If more than one hex_key preceded the MEM key, then the routine assumes that the last operation was address modification and the action of this command is to modify memory contents. The new contents of the memory are specified in the same manner as that for the registers. The chaining of this command with the NEXT command is exactly similar to the way ADDR command is chained with NEXT.

CNEXT Routine (Activated by NEXT key);

This command is used for executing another command repetititively with autoincrementing of argument. For example, if the NEXT key follows an ADDR command, the memory address is incremented by one and the new address is displayed along with its contents. When the NEXT key succeeds MEM command, in memory modification mode, the address is incremented in the same manner, thus providing a quick and easy method to modify contents of consecutive locations in memory. In register_pair display mode, the NEXT key displays the five register_pairs successively with wrap around. If the NEXT key succeeds the REG key, then the registers are displayed successively in the sequence H, L, A, B, C, D, E, F and then wraps around to display H again.

CRUN Routine (Activated by RUN key);

The function of this command is to execute a user program from either a specified location in memory or from the default address specified by user PC. The contents of the RUNM (Run_Mode Flag) is switched to zero, so that the program execution occurs continuously.

CSTEP Routine (Activated by STEP key);

The action of this command is also to execute a user program, just as RUN command does, except that the execution is done in steps of single instructions i.e. exactly one instruction is executed when the STEP key is depressed each time. This is possible by storing a one flag in the RUNM switch so that an instruction by instruction trap is set through RST 5.5. How the interrupt mechanism provides a single instruction break - point, is described in section 6.2.3.2.

CBKPT Routine (Activated by BKPT key);

This command allows setting break_point addresses and depth for debugging purposes. The break_point address is saved in BRKA and the depth parameter in BRKD of the Control Block. The command provides for both, examination of the current break_point address and depth setting as well as setting new values of these parameters. If no address parameter preceded (an address parameter results from ADDR key succeeded by two or more hex_keys) the BKPT key, this routine simply displays the current settings as per the following format;

First four digits display the contents of BRKA, the following two digits display the letters `bp' and the last two digits display the contents of BRKD.

If an address parameter preceded BKPT key, then the routine concludes that a new break_point is to be set and therefore takes the following action. The address parameter keyed_in earlier is taken as the new break_point address and is stored in BRKA. The first four digits on the HHKD module display this value. The next two digits display the letters `bp' just as in other case but the last two digits remain blank signalling to the operator that the program is awaiting the depth parameter to be keyed in. When the operator does that, the value is displayed and the last two key values are stored in BRKD. If during this sequence, CLEAR key is depressed, the breakpoint is cleared. By setting a breakpoint how debugging is facilitated, is described in section 8.1.2.

CERST Routine (Activated by ERST key);

This command simply calls the ERESET subroutine, which resets the error condition. An error condition (or state) may result from any improper operation. When an error occurs, ICMS stores an error code in ERCODE of control Block for later analysis. An error described in section 7.4). An error code `0' means no error. Hence, to reset error state, ERCODE is simply cleared. Also, the error indication lamp is turned off.

CLEAR command key simply returns the control to GETCOM and therefore has the effect of aborting the current command.

CMTEST command process routine performs RAM diagnostic and is described in section 8.1. The group of routines CRPHLD, CVECT, CRPRUN, CRPBRK, CRPSIN and CRPHLD control operation of RTP and their function is described in section 6.2.4. The command process routines CDIGX provide RTP diagnostics and their function is described in chapter 8.

The command process routines make use of a large number of sub_routines to implement primitive operations such as reading the keboard, displaying a character, fetching a byte of an argument etc. The functions of these sub routines are described in the source listing of ICMS presented in appendix

.

6.2.3. Interrupt Handling The manager's MPU may be interrupted in the following five ways;

1. Restart 7 (INTR).
2. Restart 5.5 (RST5.5)
3. Restart 6.5 (RST6.5)
4. Restart 7.5 (RST7.5)
5. TRAP

These interrupts are caused by the hardware to provide the following functionality;

6.2.3.1. RST7 Service Routine The hardware INTR signal is used for a single vector which is `/FF' corresponding to RST7. This interrupt is caused when any attempt is made to address non_existent memory. The interrupt handler routine recognizes this as an error condition and marks it down as code 6 error. Error indicator is turned on and error message is displayed on the HHKD module. Finally, control is returned to GETCOM of KHDMON, thus aborting the process which made the bad memory reference.

6.2.3.2. RST5.5 Service Routine This interrupt signal is used for setting an instruction by instruction trap so that a user program may be single stepped while being debugged. The hardware on this input is so organised that if unmasked, an RST 5.5 interrupt will be raised after 2 instructions from an ARMST5 instruction. In other words, the MPU is ready to be interrupted after it has executed two instructions (interrupting hardware is triggered on any M1_States), if ARMST5 signal, which resets the interrupting hardware, is not asserted. Therefore, during the normal operation, RST 5.5 is always masked. The only time it is unmasked is when single_step command is executed. The single_step command routine first disables the interrupts and then unmasks RST 5.5. At the end of the routine, before returning control to caller, the interrupts are enabled and just prior to that ARMST5 is asserted. The interrupting hardware is so designed that once ARMST5 is asserted, RST5.5 interrupt is not raised until two instructions later, thus allowing a return from the Command Process

Routine. When the next instruction, in user program is executed, an RST5.5 is raised. Upon entry, the interrupt handler routine again sets the mask on RST5.5 and enables the interrupts. This returns the state to normal again. At this point KHDMON is again waiting for a command key. If the STEP key is depressed once again, the same sequence will be repeated, causing execution of another instruction in the user program.

6.2.3.3. RST 6.5 Service Routine RST 6.5 provides the handshaking between ICMS and ICMON. Referring to the hardware organisation of host- processor interface (section 5.2.2.2), whenever a word is written by the host processor in the command Port of the interface, an RST 6.5 is raised. ICM manager's MPU recognizes the interrupt, suspends lower priority activities and transfers control to this service routine. The sevice routine fetches the command word's lower byte which represents the numerical value of the command. This value is used as an offset into the monitor table maintained by ICMS, which is in exact correspondence with the command table of ICMON. Hence, for every ICM related command in ICMON, there is a corresponding service routine in ICMS, as a part of the RST 6.5 Handler, which is responsible for appropriate action on behalf of the ICM manager. Thus service routine #0 maps to command #0 of ICMON and its function is to load the ICMS version and sub_version numbers into the data_port of the unibus_interface.

The synchronization between the host processor and the manager's MPU, (both machines operating at significantly different speeds) is provided mainly by software with a little help from hardware. Two hardware signals WAITIN and WAIOT are generated by MPU in the IO space to negate the "ready" input if the appropriate buffer in the Host_Processor Interface is not read /written by the host_processor. WAITIN synchronizes an Input transaction to ICM manager whereas WAITOT synchronizes an output transaction. If the Input Buffer is not_full, when WAITIN is asserted the READY input to MPU gets negated, thereby putting the MPU to sleep until the Input buffer gets full as a consequence of being written by the host processor. The moment Input Buffer Full signal goes high, READY input gets asserted and MPU awakens to resume processing. In exactly same manner, WAITOT synchronizes an output transaction by MPU, being conditioned on Output Buffer Full signal. The synchronization of the host processor is done through software by polling status bits. IBF (Input Buffer Full) and OBF (Output Buffer Fullnot) signals can be read by the Host Processor as status bits from the command port. ICMON routines test these bits in a loop before doing a read or write operation to the data_port.

6.2.3.4. RST 7.5 Service Routine This interrupt facility provides monitoring of ICM status changes caused by Color Data Formatter. The coding of status bits is explained in section 3.1 (figure 10). The inhibit bits S2, S3, S4 and S5 activate hardware in the data interface board to inhibit the corresponding colors and thus do not require any special attention. However, if any change occurs either in the op_mode bits (S7,S6) or the color_mode bits (S1, S0) the ICM manager must become aware of it. Hence, hardware circuit using four open_collector EXNOR gates, with their outputs wired_OR, detects any change in these bits. The output of this circuit drives the RST 7.5 input, which is edge triggered. Any change in these four status bits thus causes an RST7.5 interrupt. The interrupt handler routine performs the various house_keeping functions such as flags ICMBSY, signals error indicating need for initialization, as well as fetches the new status code and stores it in the control block.

6.2.3.5. TRAP Service Routine The TRAP interrupt is used for recovery from possible hang_ups by the host_processor. Every time an I/O is initiated through the data_port of Unibus_interface, a retriggerable monostable is triggered. If the host processor does not complete the transaction within a reasonable period of time, the monostable times out setting a flip_flop which raises TRAP interrupt. The TRAP service routine notes this situation as an error condition and marks it down with the code `/0A'. The error indication lamp is turned on and the error message is displayed on HHKD module before the watchdog is reset by asserting TRACK signal.

6.2.4. RTP Operate Routines Library The operation of RTP is controlled by the ICM manager through a set of routines which may be activated by depressing appropriate key on the H}HHKD module. Some of these routines can be invoked by ICMON through the RST6.5 Service Routine. The group of command process routines which perform this service are described below;

CRPCLR Routine (Activated by RCLR key);

This command simply calls the sub routine RPCLR, which asserts SYSCLR , clearing all the registers in RTP.

CVECT Routine (Activated by VECT key);

This command facilitates setting address vector for RTP in an arbitrary manner. An address parameter for RTP (one byte) is loaded in RPVECT of the Control Block either implicitly through the command RBRK or explicitly by the operator accessing RPVECT just as any other memory location. The address parameter thus stored is used for vectoring the RTP. However, some arithmetic is also performed to compute a new address parameter in anticipation of being used for subsequent vectoring operations. In case of break_point runs, it is desirable to keep track of the break_points. If the break_point depth is non_zero, an offset equal to break point depth is added to the current vector address and stored back in RPVECT after the current address vector is set in the RTP. Thus the new contents of RPVECT indicates simply the next breakpoint. If the break_point depth is set to zero, then the address parameter is decremented by one and stored back in RPVECT. This allows the possibility to go backwards to any point in the micro instruction stream by successively depressing the VECT key. To go forward, the break_point depth may simply be set to `/01' or some other non_zero value as the operator may desire. The two possibilities on address arithmetic provide a complete control on setting any address vector for RTP. The RTP's microprogram address register is co erced to the specified vector by the sub routine VECTOR in the following manner; The address parameter is picked by VECTOR sub routine from RPVECT and set as the vector address input to RTP's microprogram controller through RTP interface. Then an interrupt is caused to RTP. The effect of the interrupt is to jam a JUMP_ZERO instruction to the microprogram controller. By convention, a JUMP_VECTOR instruction is always stored in location zero of the microprogram memory. Thus by single stepping RTP exactly three times (which is necessary because of two_level pipelining), the vector address is transferred to the microprogram address register.

CRPBRK Routine (Activated by RBRK key);

This command allows examination of the current vector address parameter (contents of RPVECT) and the break_point depth parameter (contents of RPBRKD) as well as provides the ability to change these parameters arbitrarily. If no address parameter preceeds this key, the current contents of RPVECT and RPBRKD are displayed. If an address parameter (only one byte addresses for RTP) preceeds this key, the address parameter is stored in RPVECT and is displayed. The breakpoint depth field is blanked at this point however, to signal to the operator that the depth parameter is being awaited to be keyed in. When the operator responds, the depth parameter is stored in RPBRKD.

CRPRUN Routine (Activated by RRUN key);

This command sets RTP in RUN mode in the following manner; If the break_point depth is set to zero, sub_routine RPRUN is simply called to set the RUN mode (RTPRUN = HIGH) after making sure that other control signals are properly initialized. If the break_point depth is set to a non_zero value, then the RTP is single_stepped through the specified depth by calling RPSING successively.

CRPSIN Routine (Activated by RSIN key);

This command calls RPSING to single_step RTP. The microprogram address register as well as the contents of RTP's Bus_A and Bus_B are displayed as per the following format; First two digits display the microprogram address register (equivalent of program counter). The next digit is blanked followed by two digits displaying bus_A, followed by another blank and then the last two digits displaying bus_B.

CRPHLD Routine (Activated by RHLD key);

This command simply calls the RPHOLD sub_routine which halts RTP by setting the RTPRUN signal low.

6.2.5. RTP Diagnostic Routines Library A variety of diagnostics can be performed on RTP by a set of routines, eight of which may be invoked from the keyboard of the HHKD module. These commands are called DIG0 through DIG7 and their operation is described in chapter 8.

6.3. ICM Support Software

This section describes the group of programs which are required ICM in addition to the support provided by the local UNIX system, to fulfil the needs of ICM.

6.3.1. Intel Hex-Formatter (INHEX)

This program formats an absolute download module generated by UNIX command "reldld" into Intel's Hex_Format. The program was written in C_ language, the source copy of which is in the filename INHEX.C. Appendix G presents the INTEL's HEX_FORMAT and the source listing of INHEX.C. The executable code is under filename INHEX. The program operates on standard input and generates standard output. Hence pipes must be used to read from an input file and write to an output file.

6.3.2. Hex to Binary Converter (X2BN) This program, also written in C_Language, converts hex_characters into binary. The source copy of this program is in X2BN.C, a listing of which is presented in Appendix H. The executable code is in X2BN.

6.3.3. RTP Micro Assembler (MICRASS) This micro_assembler is intended for translating assembly language (perhaps mnemonic machine language) programs into machine code for the real_time Processor. Development of this micro_assembler has been the objective of an undergraduate thesis research by another student, Richard F. Makino. Although a final version is still to be produced, a first_cut version, written in C. language, has already been produced, demonstrating the viability of this approach.

CHAPTER 7

USER's View

This chapter describes operator's interaction with ICM. The primary user interface is through the host_processor. This interface is designed in a a way such that non_engineering personnel can also communicate with ICM easily. The HHKD module interface is designed to be used by engineering personnel who possess adequate knowledge of ICM hardware as well as software.

7.1. Operation via Host Processor

The operation of ICM via host_processor is provided by ICMON through the system's console terminal. ICMON may be invoked at the system command level to start communication with ICM. The system responds by printing the header with ICMON's version numbers and ICMS version numbers, followed by a prompt message. The ICMON's prompt message is "COMMAND:=", indicating that the system is awaiting user response. The user then communicates through a set of commands. Although, a small set of commands are currently supported, these provide adequate user interaction. More commands can easily simply be added upto a total of 256. Each provides a unique function as follows;

1. HELP Command;

   In response to this command, ICMON prints a list of currently supported commands.

2. GET Command;

This command is used to transfer an arbitrary length record from ICM to the host processor. The Command requires three arguments to be specified by the operator, viz. (i) Source Address (refered to ICM) (ii) Destination Address (referred to host- processor) and (iii) record length. The arguments must be typed in when ICMON asks for it. All arguments must be specified in octal. If the operator includes any illegal character, the system rejects that argument and sets its default value, which is zero. However, if an illegal character is accidently typed, the operator can undo it by continuing to type the correct number from very start, recognizing the fact that the program would accept only the last 6 octal characters for a number input.

3. PUT Command;

This command is used to transfer an arbitrary length record from the host_processor to ICM. The operation of this command is exactly the same as GET command except that the data transfer occurs in the opposite direction.

4. EDMP (Error Dump) Command;

This command fetches the error code from ICM and prints an error message on the console terminal corresponding to the error code.

5. ERESET (Error Reset) Command;

This command invokes the ERESET sub-routine in ICMS to reset the error condition. The result is same as that caused by depressing ERST key on the HHKD module.

6. RUN Command;

This command invokes RPRUN sub-routine in ICMS to set RTP in RUN mode. The result is the same as that caused by depressing RPRUN key on the HHKD module.

7. HOLD Command;

This command invokes RPHOLD sub-routine to halt RTP. The result is the same as that caused by depressing RPHOLD key on the HHKD module.

8. SDMP (Status Dump) Command;

This command fetches the status byte from ICM Control Block, decodes it and frints messages to indicate the op_mode, color_mode and color_inhibit conditions, if any.

9. INIT Command;

This command performs a complete initialization of RTP from a power_up state and then sets the RTP in RUN mode. At first, status is fetched from ICM and decoded for color_mode. Then appropriate micropro-gram and Ink_Correction Table (if required) are down-loaded in RTP through the ICM manager. Finally, the RTP is set in RUN mode. Thus, this single command brings up the system completely automatically and the operator is not burdened with details.

10. DIGx (Diagnostic) Commands;

These commands perform diagnostic on ICM. DIGO through DIG7 cause exactly the same diagnostic operations as that caused by identically labelled command keys on the HHKD module. The exact operation of these 8 diagnostic commands is described in chapter 8. However, diagnostic operations through this interface is not limited to only these 8 routines. More of these commands (upeasily to a total of) can be added to implement user defined diagnostics.

11. ODMP (Octal Dump) Command;

This command does not involve any interaction with ICM. Since the system was initially debugged on a bare machine (PDP-11/10), such utilities had to be locally created. As the name implies; this command provides host_processor's memory dump in octal. It requires two arguments viz. (i) the source address and (ii) a count of words to be dumped.

12. LOAD Command;

Like ODMP, LOAD provides another utility function and does not involve any interaction with ICM. Using this command a user is able to load user specified data at any location in the host_processors memory from the console terminal. This command also takes two arguments viz. (i) Destination Address and (ii) Word Count.

13. CMPR Command;

CMPR provides another utility function and does not interact with ICM. With this command, two equal length records in memory may be compared for equality. This command requires three arguments to be specified viz. (i) Source Address (start address of record 1), (ii) Destination Address (start address of record 2) and (iii) Word Count (record length). In case of equality, an appropriate message is printed. If the contents of the memory locations did not match during a comparision, then the source address and its contents as well as destination address and its contents are printed. Finally, the tally of comparisions which did not procuce a match, is printed.

14. LDPT (Lood Paper Tape) Command;

This command provides yet another utility function. The operation is similar to LOAD command except that the input comes from the paper tape reader of the console terminal instead of keyboard of the console terminal.

15. EXIT Command;

This command terminates conversation with ICMON and returns the control to the host operating system.

7.2. Operation Via HHKD Module

The HHKD module interface provides extensive system_wide control over ICM. Since the hardware of RTP is completely under the control of manager's software, it is possible to do anything through this interface. The function of each command key has already been described in chapter 6.

One approach to using this interface is to go through a keying sequence such that the manager executes the desired function using the built_in commands.

Another approach may be to write a program in 8085_executable_code and load it in the local RAM of the ICM manager. The control then may be transferred to this code segment in the following manner; Depress ADDR key and set_up the start address of the user program by keying_in appropriate hex_characters. Finally, depress RUN key (not RPRUN) for executing the user program. However, this approach is only recommended for experts, who understand the details of hardware and software design of ICM, since improper I/O addressing may cause hardware damage.

The possibilities of using this interface is innumerable. As such, it is not possible to set a guideline for how this interface should be used. Rather the functionality of each command key has been described so that a user may derive the desired result by working out an appropriate combination.

In order to attach the HHKD-module, the system need not be powered down. However, the power switch must be off (indicated by +5v LED not glowing) while the cable is physically being attached.

7.3. ICM Data Bases

The ICM requires two sets of data bases to be supported by the file system of the host_processor. These data bases contain the microprograms and the Ink Correction tables required for downloading the RTP memories. The data bases may be prepared in the environment of the host processor or transported from elsewhere. The following organisation is recommended although user may adapt a different approach for his own convenience;

7.3.1. Ink-Correction Microprogram Library. This library contains a total of four microprograms for real_time operation of ICM and additionally may contain any number of diagnostic microprograms. The real_time microprogram are named as follows;

1. mpgm.key
2. mpgm.ylw
3. mpgm.cyn
4. mpgm.mgn

Each of these programs can be maximally 770 (octal) words long, although in actuality, the programs are much shorter than that. The current version of ICMON (which was designed for the DEBUG environment and may differ considerably in this respect from the version finally used for the operational environment) assumes that these microprograms are somehow loaded in the main memory of the host processor as contiguous 770 (octal) words long records.

The eight diagnostic microprograms currently supported by the system, live in the program memory of the ICM manager. However, it is foreseen that, in future, more diagnostic programs will be added to the system. Those can also be be supported under this microprogram library.

7.3.2. Ink-Correction Table The other data base required for real_time operation of ICM contains the Ink Correction tables for the various color modes. These are named as follows;

1. ict.ylw
2. ict.cyn
3. ict.mgn

Each of these data records are exactly 1K words long. Table 1 specifies the format in which these records must be organised. As in the case of micro_ programs, the present version of ICMON assumes that these records are loaded as contiguous records in the main memory of the host_processor.

7.4. Program Development Environments

Development of software for ICM requires system support at the following three levels;

7.4.1. Microprogramming for RTP Microprograms may be written for RTP directly in hexadecimal machine language. Every microinstruction must be eight bytes in length and conform to format specified in appendix D. Appendix J presents some microprograms written in Hex machine language. A translator must then be available to translate the program from hexadecimal to binary. The utility program X2BN, described in section 6.3.2, provides exactly this function.

An alternative to programming_in_hex is to use some sort of a symbolic language or mnemonic machine language. Appendix K presents a brief summary of RTP instruction set and the syntax of a mnemonic machine language which should make programming much easier. Appendix L presents a model microprogram written in this mnemonic machine language. However, in order to generate binary object code for actual downloading, a micro_assembler is required. MICRASS, the micro_assembler for RTP, described in section 6.3.3. provides this capability. Although, MICRASS is currently under development [ ], sufficient work has already been done to demonstrate the viability of this approach.

7.4.2. Programming ICM Manager. In order to develop programs for the ICM manager, basically an assembler for 8085-code is required to be supported by the system. The CIPG UNIX supports a cross-assembler viz. MICAL (Microprocessor Cross-Assembler) which generates code for Intel's 8080 microprocessor. MICAL may be used for assembly of 8085 code also because the instructions sets of the two machines are completely identical except that 8085 implements two more instructions viz. RIM and SIM. To use RIM and SIM instructions in the source program, and still be able to use MICAL for assembly, one easy way out is to define RIM, SIM as global symbols and equate them to their opcodes.

The MICAL assembled code on UNIX is a relocatable object module. However, UNIX conveniently supports a shell-level command 'reldld' which generates absolute download modules from relocatable object modules such as one generated by MICAL. If the program is to be located in the EPROMs (2708's), then it is required to be punched on paper tape in an appropriate format, so that it can be transported to a programming equipment. During the course of development of ICMS, the programming equipment used was the INTELLEC MDS system installed in the Digital System Laboratory. The MDS system supports a universal PROM programmer system on which EPROMs (type 2708's) can be conveniently programmed. However, in order to program the EPROMs on MDS equipment, it is necessary that programs are punched on paper tape in Intel's hex_format (appendix G.1). Therefore, the system must support a utility program to convert absolute download modules (output of "reldld" command) to Intel's hex_format. The program INHEX, described in section 6.3.1.,was developed for this purpose.

<u>7.4.3. Programming Host Processor</u> The program development environment required for host_processor perhaps needs no explanation. However, in the context of debug_environment, a brief description of the available system support follows; The

host_processor used during ICM's debugging_phase was a PDP-11/10. This was a bare machine, devoid of any software except for a bootstrap routine in firmware. A vendor supplied absolute loader was available on paper tape. Earlier researchers faced with the problem of using this bare machine, created a system macro called "absload" which provides assembly of MACRO-11 source code as well as binding of the relocatable module to the specified absolute address and finally, the resulting code punched out on paper tape. Thus, invocation of "absload" command on the first argument <source_filename> and the second argument <absolute-address-of-program- origin>, produces a paper tape which can be directly read on PDP-11/10. This was the approach taken to load ICMON in the PDP-11/10. ICMON provides a completely stand_alone operation. Hence, once loaded and started, it provides all the utilities necessary for carrying out ICM related operation.

CHAPTER 8

Diagnostics

Extensive diagnostic features have been designed into the ICM system in order to monitor the correctness of its operation as well as detect and localize faults in case of failure, as described below;

8.1. RAM Diagnostics

This feature provides an automatic testing of all RAM modules (both manager's local RAM and RTP_shared RAM) with the help of software. In case of flaky chips, it also helps to identify the defective locations. A key on the HHKD_module called RTST (RAM test) invokes the CMTEST command process routine (within ICMS) which performs the RAM testing. CMTEST requires two arguments viz. (i) start-address and (ii) End-address. These arguments must be passed to CMTEST via the register_pairs BC and DE. Register_pair DE should contain the start address and the register_pair BC, the end_address. Registers may be initialized by getting into register_display mode, by depressing the REG key on the HHKD-module. The contents of the registors may then be modified.

Every cell of the memory segment, between the two specified limits, is checked repetitively for correctness of read and write operations. Basically three types of checks are made. Every test begins at the start_address (inclusive) and proceeds through each location successively upto the end_address (exclusive). The procedure for testing is as follows; First the pointed location in memory is cleared and then checked if it indeed cleared. Next, a `1' is stored in the LSB cell of the location and checked again. The `1' bit is then shifted to the next significant bit position, stored and then checked once again. This procedure is continued until the `1' shifts out of the MSB position. Finally, `/FF' (all_ones) bit pattern is stored in the current location and checked once again before incrementing the pointer to the next location for testing. When all locations are checked in this manner, another pass is made to check if all the locations within the defined segment contain `/FF' bit pattern. If they do, the result of the test is taken to be successful, therefore, message "Good" is displayed and the test is started all over again. If during any comparision the contents of the memory did not match the contents of the accumulator (the data to be stored), further testing is halted and the address (where the operation failed) is displayed along with its actual contents as well as what the contents should have been. At this point the program waits for the operator's response. If the operator depresses "RAMTST" key again, then the testing is resumed from the brake point. If any other key is depressed, the test is started right from begining. Thus, once invoked, this diagnostic routine keeps checking RAM in an infinite loop. The only way to exit from this routine is through RESET. The program is designed in this manner so that a proper initialization of RAM is forced after this test which may have destroyed the system stack, control Block etc.

Before the test is started, the program checks if the specified test space overlays the system stack. If it does, the system stack is relocated (at the other end of the local RAM) since the diagnostic program itself uses the stack. Therefore, the lowest 32 bytes of local RAM must be excluded from the test space, if it overlays the system stack.

8.2. Manager Single-Stepping

Provision for single_stepping Program execution appears to be an essential feature on almost any machine. ICM manager's single-stepping capability provides a very powerful software debugging tool. In fact, it was possible to debug parts of ICMS itself using the single_step facility. A DIP_switch located on the manager board determines whether the manager would operate in 'RUN' mode or the single_step mode. In single_step mode, the program executes one instruction every time the 'step' key is depressed. The result of instruction execution is visible through examination of memory, registers, stack and other pointed locations. A user program (8085_code) may be loaded in manager's local RAM and debugged in this manner.

8.3. Manager Break-Point Runs

Another useful debugging tool provided by the system is the ability to set breakpoints while executing manager programs. Breakpoints may be set by specifying an address (by depressing ADDR key and following up with appropriate hex keys) and depressing the "BKPT" key. Another parameter required to be set is the break_point depth, which may be done by following the "BKPT" key with appropriate hex_keys If the breakpoint depth is set to zero, the program execution is halted the very first time Program Counter matches the breakpoint address. To resume program execution, `RUN' key may be depressed again whereupon the next occurrance of the same address will introduce another break_point. If the break_point depth is set to a non_zero value, then the program execution is halted after an equal number of occurrences of the break_point address in the program flow. This provides an useful capability of executing a program loop a given number of times (by setting the break_point within the loop) before the break_point is introduced. The introduction of breakpoint during program execution allows examination of registers, stack and as well as other memory locations thus providing the ability to monitor the execution of a program dynamically.

8.4. RTP Single Stepping

The single_stepping of the real_time processor provides similar type of capability, although in a slightly different manner. Since RTP executes micro instructions, there is no need for setting instruction cycle traps as in the case of the manager. Instead, if the central clock (WRCLK and CTLCLK) of the RTP is controlled such that it is pulsed a single time when a key on the HHKD module is depressed, the single_stepping capability would be obtained. This is exactly the manner in which RTP is designed. When `RSIN' key is depressed on the HHKD module, the 6.2 5 MHz base clock is cut_off to the colck generator and instead the clock input is pulsed a single time. Since all operations within RTP is synchronized to this base clock, the effect of the single clock pulse is to step through the micro instruction sequence just once.

8.5. RTP Vectoring

The microprogram memory of the RTP may contain more than one program module and it may be desirabe to select one of these programs for execution during a given time . This capability is particularly attractive from the point of view of on line diagnostics. However, to provide this type of operation, the manager must be able to steer the RTP to the appropriate program segment. ICM manager is able to do this through the RTP_interrupt facility as described in section 5.1.5.8. The manager puts out an appropriate address on the VECTOR port of the RTP_Interface and then causes an interrupt to the RTP. RTP responds by aborting its current task and diverting to the newly specified task. However, causing an interrupt in this manner does not save the context of the current task, which may be desirable in some instance. An alternative method for task switching through a handshake protocol is provided as described below;

A flag (RTPJMP) from the manager can be tested by RTP to make a decision about changing over to a new task, as opposed to being co erced into it through the interrupt facility. As before, the manager places the address of the new task on the VECTOR port, before asserting the RTPJMP flag. A bit from RTP's micro_code is used as a flag (RTPACK) which may be tested by the manager. When RTP finishes execution of the newly specified task, it asserts the acknowledge flag RTPACK and then may return to the execution of its earlier task or probe RTPJMP flag again to find out what must be done next. The manager, on the otherhand may test the RTPACK flag at its discretion, recognize that the requested task has been serviced and may proceed to specify another new task.

8.6. RTP Break-point Runs

The breakpoint run facility for RTP is designed in a somewhat unconventional manner for the sake of simplicity. An operator may specify a microprogram address (one byte only) and then specify a breakpoint depth by depressing appropriate hex-keys following the `RPBRK' key. Next the operator must depress the `VECT' key to load the microprogram address register with the specified address. Now, if the `RRUN' key is depressed, RTP would execute the microprogram starting at a address through the specified number of microcycles specified by the depth field. If "RRUN" is depressed again the microprogram execution resumes from the broken point but gets suspended again after the specified depth.

8.7. Diagnostics with Test Board

A number of diagnostic functions can be performed on RTP with the help of a test board shown photographically in appindix B. The test board is required to be plugged in the place of the Data_Interface board (#9). The test board carries 3 sets of 8_bit dip switches for emulating the input data. One 8_bit dip switch is used for status information and two hex_displays always display the output register OR3. Thus, this test board provides a simple I/O interface for the real_time processor. Diagnostics are performed with the help of pre_written diagnostic routines. Currently 8 such diagnostics routines are supported, which may be invoked either by depressing keys on the HHKD module or by typing the command on the host-processor's console terminal. A copy of all the diagnostic micro_routines are stored in the manager's program memory. When invoked, the manager downloads the corresponding routine into the microprogram memory of RTP and sets it in run mode to execute the routine. The operator may set any arbitrary input on the test board and compare the displayed output with a predetermined reference to establish whether RTP generates the right result.

8.7.1. Machine Dependent Diagnostics Commands DIG0, DIG1, DIG2 and DIG3 perform machine_dependent diagnostics in the following manner.

8.7.1.1. Diagnostic Routine DIG0; This routine performs testing of the ROSH element of RTP's ALE. The input from IR1 is shifted left eight times in word_mode and the MSB from ROSH is moved to OR3. Thus the displayed output same as the input. Appendix J.1 presents the mecroprogram for DIG0 in Hex.

8.7.1.2. Diagnostic Routine DIG1; This routine performs testing of the ALU component of RTP's ALE. The input from IR1 is added to IR2 and the result is clamped for overflow before being moved to OR3. Appendix J.2 presents the microprogram for DIG1 in Hex. 8.7.1.3 Diagnostic Routine DIG2; This routine performs testing of the MAC component of RTP's ALE. The input from IR1 is multiplied with that of IR2 and the result is subtracted from IR3. The final result is moved to OR3. Appendix J.3 presents the microprogram for DIG3 in Hex.

8.7.1.4. Diagnostic Routine DIG3 This routine performs testing of real_time decision making by executing an algorithm which selects least of three numbers. The three numbers are input from IR1, IR2, IR3 and the selected number, having the least value, is moved to OR3. Appendix J.4 presents the microprogram for DIG3 in Hex.

8.7.2. Application_dependent Diagnostics Commands DIG4, DIG5, DIG6 and DIG7 perform application_dependent diagnostics. The algorithm selected for this purpose is the same as described in section 2.8, which is to compute the approximations $\bar{Y}$, $\bar{C}$, $\bar{M}$ and K. The inputs R,G, B are taken from the test board via IR1, IR2 and IR3 respectively. The result is moved to OR3 and displayed on the test board.

8.7.2.1. Diagnostic Routine DIG4; This routine performs computation of the Yellow_approximation as per the following equation;

$$\bar{Y} = 1.8671875 * D_R - 0.5703125 * D_G - 0.109375 * D_B$$

Appendix J.5 presents the microprogram for DIG4 in Hex.

8.7.2.2. Diagnostic Routine DIG5; This routine performs computation of the Cyan_approximation as per the following equation;

$$\bar{C} = -1.15625 * D_R + 1.9921875 * D_G - 0.2734375 * D_B$$

Appendix J.6 presents the microprogram for DIG 5 in Hex.

8.7.2.3. Diagnostic Routine DIG6;

This routine performs computation of the Magenta_approximation as per the following equation;

$$\overline{M} = 0.1875 * D_R + 0.3671875 * D_G + 0.6328125 * D_B$$

Appendix J.7 presents the microprogram for DIG6 in Hex.

8.7.2.4. Diagnostic Routine DIG7; This routine performs computation of the UCR-Component (Key) as per the following equation;

$$K = 0.5 * [\text{MIN} \{ \overline{Y}, \overline{C}, \overline{M} \}]$$

For the sake of clarity of understanding, a simple MUCR is chosen.

Appendix J.8 presents the microprogram for DIG 7 in Hex.

CHAPTER 9

Bench_Marking and Performance Evaluation

Although it may not be proper, in some sense, to compare the performance of ICM's RTP with any contemporary off_the_shelf microprocessor, yet, comparisions have to be mode, in the absence of any other yardstick for performance evaluation. Intel's 8086_family microprocessor was chosen for the comparision since that came closest to the needs of this application, as described in section 1.5. The bench_marking algorithm was chosen to be the Ink Correction algorithm. While evaluating the performance of 8086_based system, it was assumed that I/O is architectured with Parallel Interface chips 8255As, such that maximum thruput is derived. It was estimated that the time required to compute the ink_densities from R, G, B inputs corresponding to one picture_element would be of the order of 400 microsecs, with 8086 being driven at 6.25 MHz clock frequency. To do the same job the RTP takes less than 11 mecrosecs, operating at the same clock frequency.

CHAPTER 10

Future expansion and other Applications

It may be desirable (for some other application), to add a few more hardware capabilities to the real_time processor. Provision exists for adding a hardware stack which would enhance the capabilities of RTP significantly. Provision also exists for adding a hardware queue, which would support convolution type computation very efficiently. Board space (on board # , columns) as well as spare bits in micro-code are available for these hardware additions. However, due to lack of time, these elements could not be implemented (Implementation of these elements was given lower priority since these were not essential to the ICM application).

In respect of software, it may be desirable to expand the diagnostic section some more. It was thought that more diagnostics would be added by the users of the system. Another potential area for future expansion is the ICM manager. With very little addition of hardware, the manager's (8085's) serial communication can be developed so that, with appropriate software, the manager can directly communicate with a terminal. This way, it is possible to architecture a completely stand_alone system, eleminating the need for interaction with a host_processor (ofcourse in the case of this application, the host_processor has many other functions).

Even though ICM was developed for a specific application, it has the flavour of a general_purpose computing machine. The same machine can be used for many other applications requiring perhaps very little hardware change (may be in the I/O area). The Real_time processor can be made to execute a variety of tasks simply by swapping microprograms. It is the opinion of the author that this type of machine can be an excellent tool for real_time speech processing or facsimile_speed image processing.

CHAPTER 11

Conclusions

A specially architectured microprogrammable processor, capable of high_speed computation to match engraving system thruput, has been developed. The real_time processor can compute the ink_densities from R, G, B ICI values within the allocated time slice of 11.5 microsecs. The RTP is managed by a resident 8085-MPU based microcomputer through software, yielding a highly flexible, yet computationally powerful system. High_speed inter_processor communication with a larger system (PDP-11 based) is provided. Presence of a microcomputer as the system manager provides extensive testability and diagnostic capabilities. Both hardware and software systems have been designed in a well structured manner yielding high degree of modularity. The Ink-Correction module is not only able to do the real_time image processing for the Helio_Klischograph, but can serve as a development tool for any other real_time signal -processing application as well.

BIBLIOGRAPHY

1. Parks, E.L., "New Generations of Electronic Scanning and Engraving Equipment", Proceedings of the Technical Association of the Graphic Arts, 1973, pp. 105-115.

2. Troxel, D.E. & Schreiber, W.F. et al. "Automated Engraving of Gravure Cylinders", paper presented at the Symposium on Digital Systems, Tenth Anniversary, Simon Bolivar University, Venezuela, March 1980.

3. Berberian, H.A. "Error Reduction in a linear Transformation Color Correction Scheme", MIT/CIPG memo PROV-40, January 1980.

4. Berberian, H.A. "Determination of Look_Up_Table Entries for the Ink Correction Module", MIT/CIPG memo PROV-56, July 1980.

5. Berberian, H.A. "Smoothing of Color Matching Data for the Ink Correction Module", MIT/CIPG memo PROV-57, July 1980.

6. Schreiber, W.F. "Look Up Table for Color Correction", MIT/CIPG memo PROV-51, June 1980.

7. Schreiber, W.F. "Obtaining Look_Up_Table Data from Smoothed Matching Data", MIT/CIPG memo PROV-54, June 1980.

8. Troxel, D.E. & Schreiber, W.F. et al. "Bandwidth

compression of High Quality Images", paper published in International Conference on Communications, IEEE, June 1980.

9. Wyszecki,G. and Stiles,W.S. "Color Science, Concepts and Methods, Quantitative Data and Formulas", John Wiley & sons, Inc., 1967.

10. Mishra,S.N. "Scaling and Quantization of Matrix Co-efficients for Implementation on ICM Hardware", MIT/CIPG memo PROV-60, August 1980.

11. Buckley, R.R. "M_dimensional Piecewise Linear Interpolation", MIT/CIPG memo PROV-13, March 1979.

12. Schreiber, W.F. and Troxel, D.E. "Color Processing for the Helio_Klischograph, MIT/CIPG memo, July, 1978.

13. Yule, J.A.C. "Principles of Color Reproduction", John Wiley & sons, Inc., 1967.

14. Buckley, R.R. "Reproducing Pictures with Non-reproducible Colors", S.M. Thesis, Department of Electrical Engineering and Computer Science, MIT., 1978.

15. Salisbury, A.B. "Microprogrammable Computer Architectures", American Elsevier Publishing Co., Inc., 1976.

16. Hwang, Kai "Computer Arithmetic: Principles, Architecture and Design", John Wiley & Sons, Inc., 1979.

17. Klingman, Edwin E. "Microprocessor Systems Design",

Prentice_Hall, Inc., 1977.

18. Artwick, Bruce A. "Microcomputer Interfacing", Prentice_Hall, Inc., 1980.

19. Eckhouse, R.H.Jr. "Minicomputer Systems: Organization and Programming (PDP-11)", Prentice_Hall, Inc., 1975.

20. Fletcher, William I., "An Engineering Approach To Digital Design", Prentice_Hall, Inc., 1980.

21. Intel Corporation: MCS_85 User's Manual.

22. Intel Corporation: 8080/8085 Assembly Language Programming Manual.

23. Intel Corporation: The 8086 Fammily User's Manual.

24. Intel Corporation: MCS_86 Assembly Language Reference Manual.

25. Advanced Micro Devices: Build A Microcomputer Series, 1978.

26. Advanced Micro Devices: The Am 2900 Family Data Book.

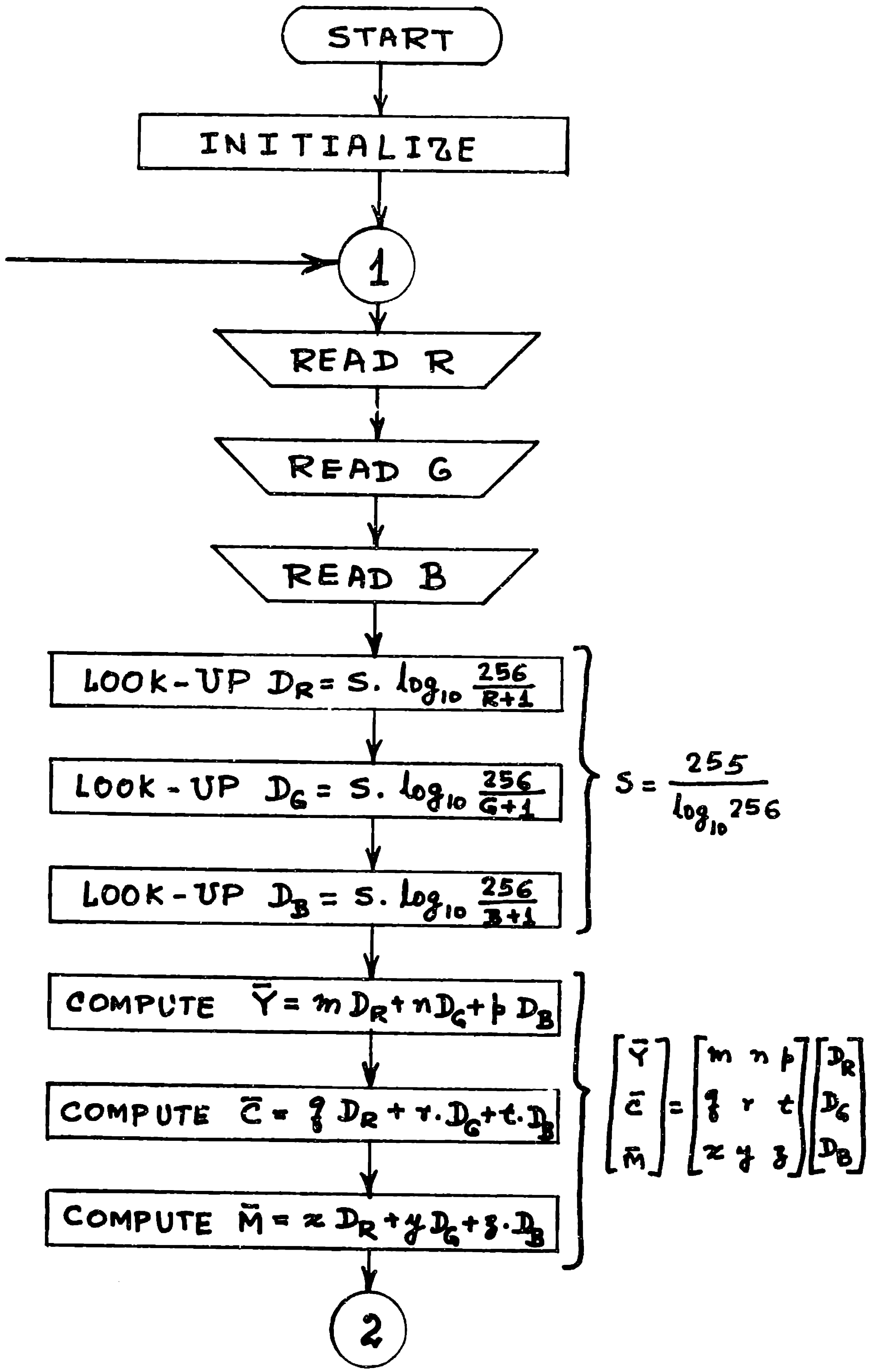
27. Fairchild: Bipolar Memory Data Book.

28. National Semiconductors: TTL Data Book.

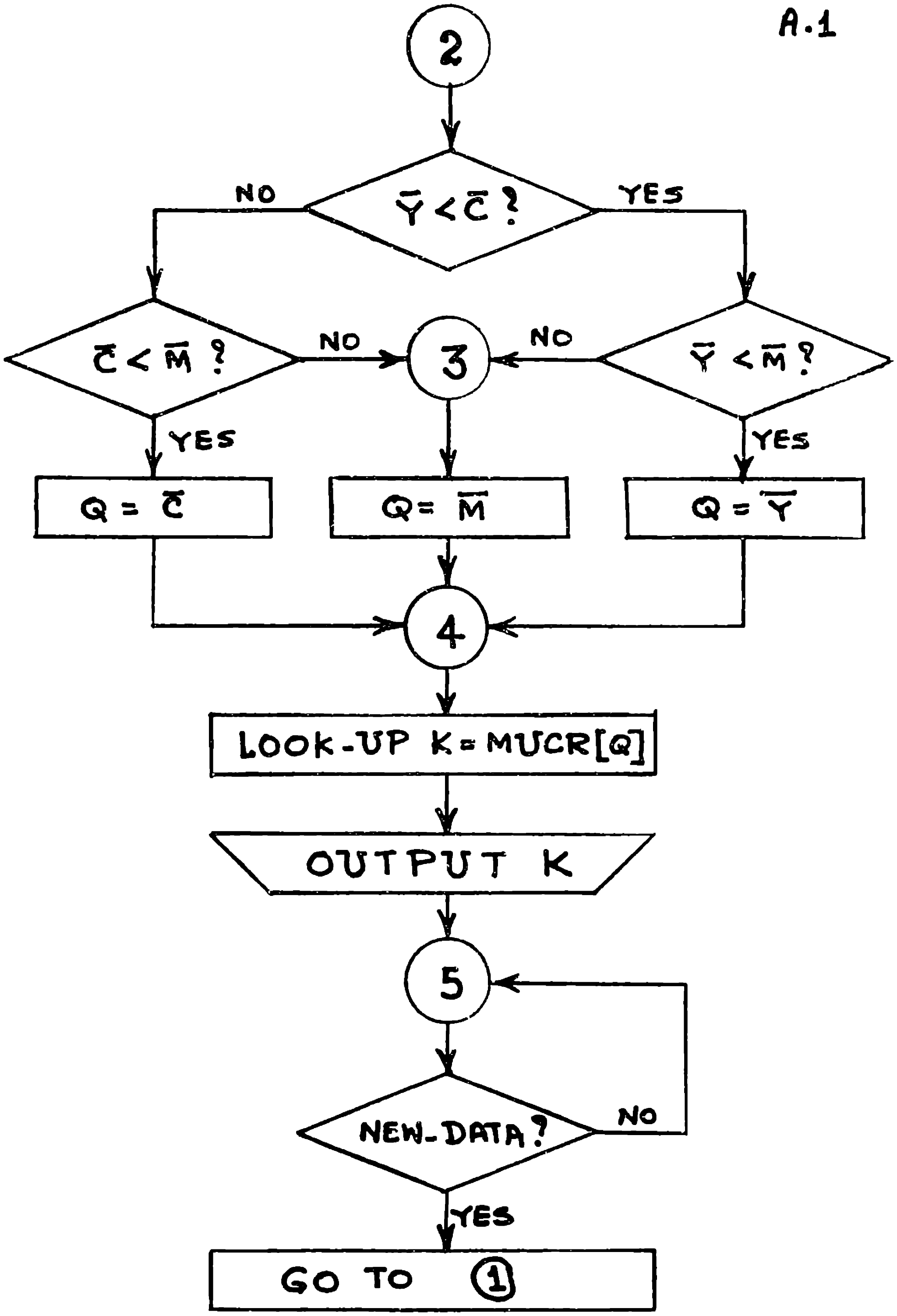
29. Texas Instruments: TTL Data Book, 2nd Edition.

30. Texas Instruments: Low Power Schottky and Advanced Low Power Schottky Products.

MACRO-FLOW CHART FOR K



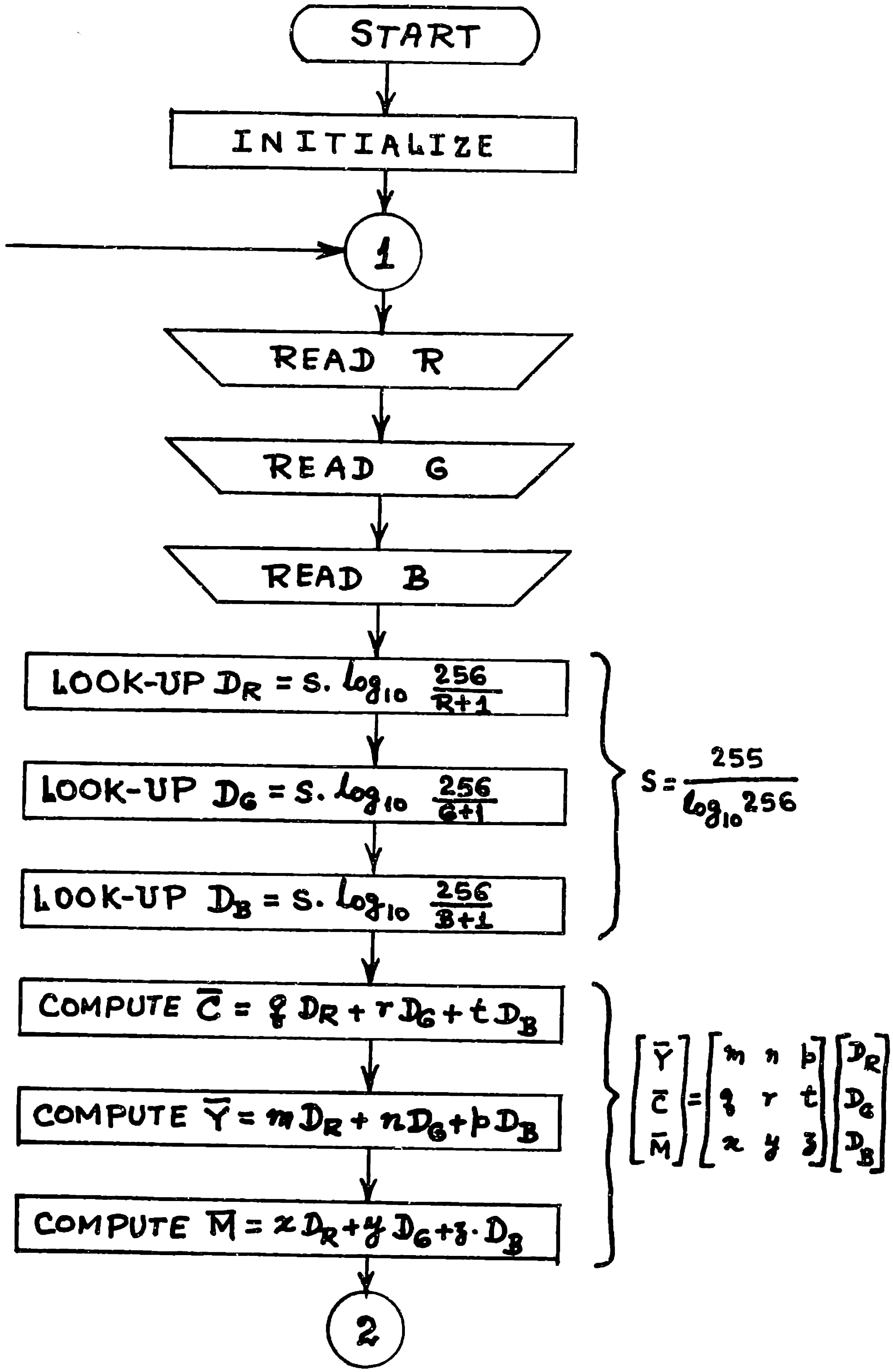
A.1

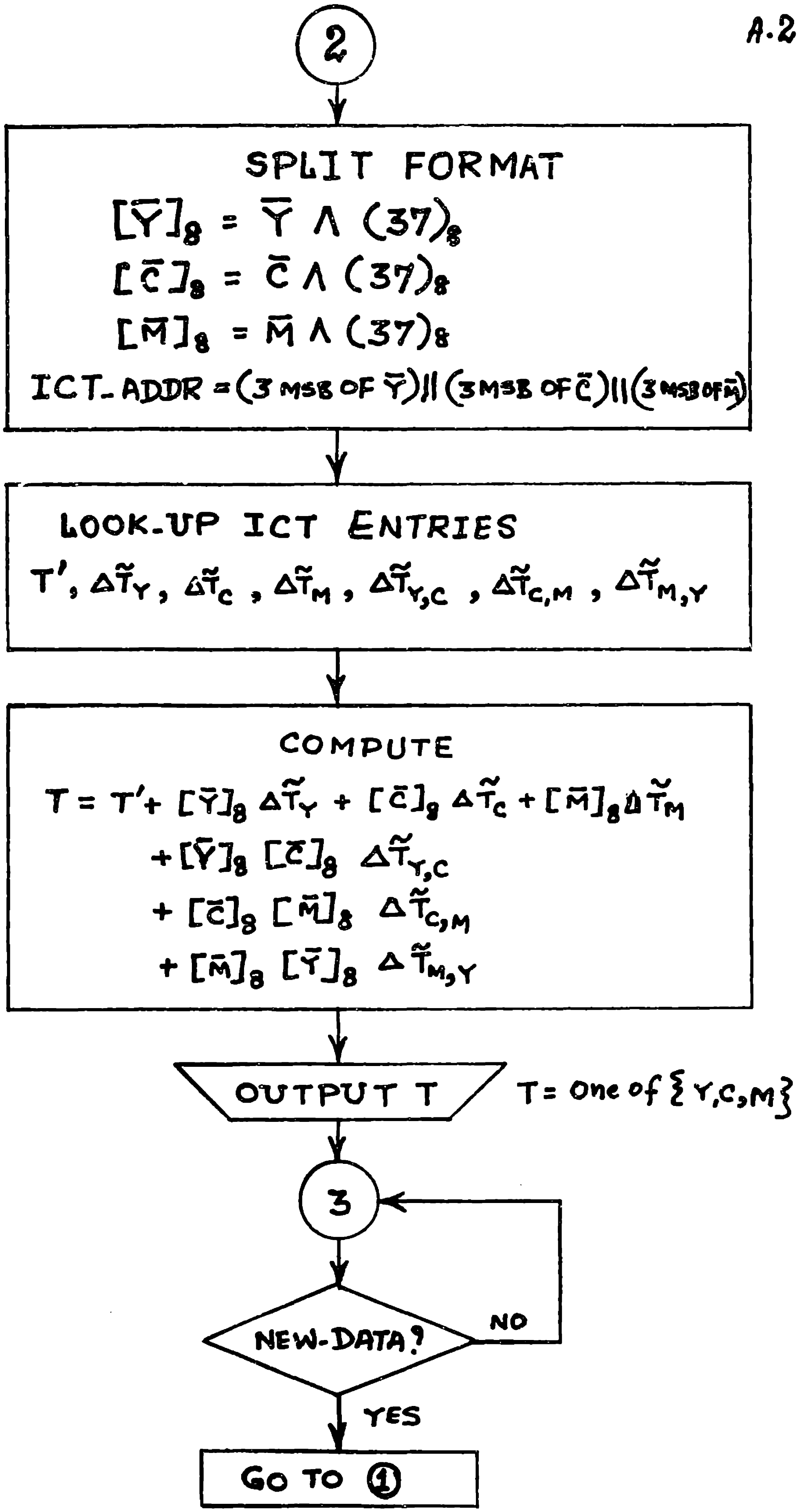


2
$\bar{Y} < \bar{C}$?
NO
YES
$\bar{C} < \bar{M}$?
NO
3
NO
$\bar{Y} < \bar{M}$?
YES
YES
$Q = \bar{C}$
$Q = \bar{M}$
$Q = \bar{Y}$
4
LOOK-UP K = MUCR[Q]
OUTPUT K
5
NEW_DATA?
NO
YES
GO TO 1

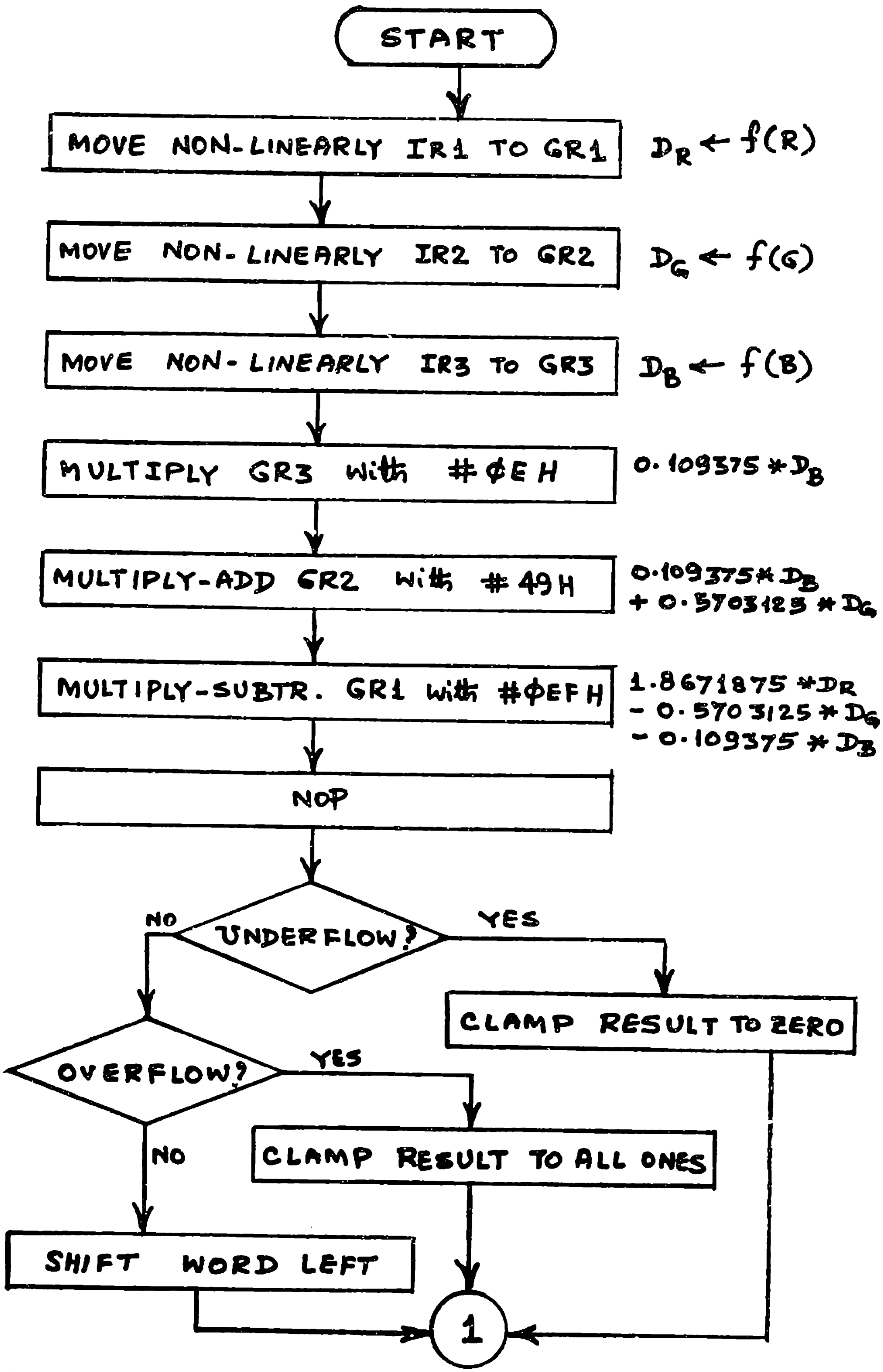
APPENDIX-A.2

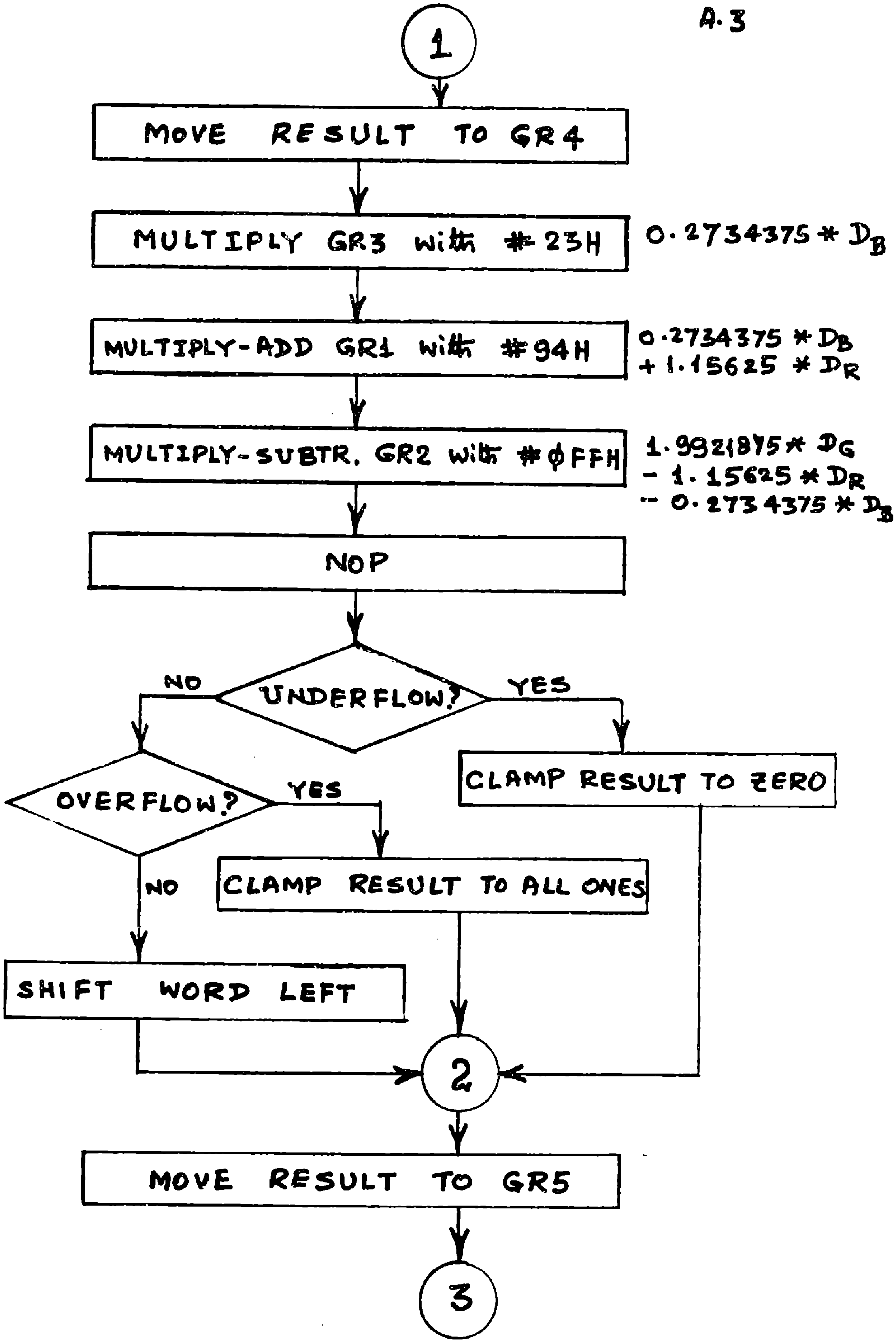
MACRO-FLOWCHART FOR Y, C, M

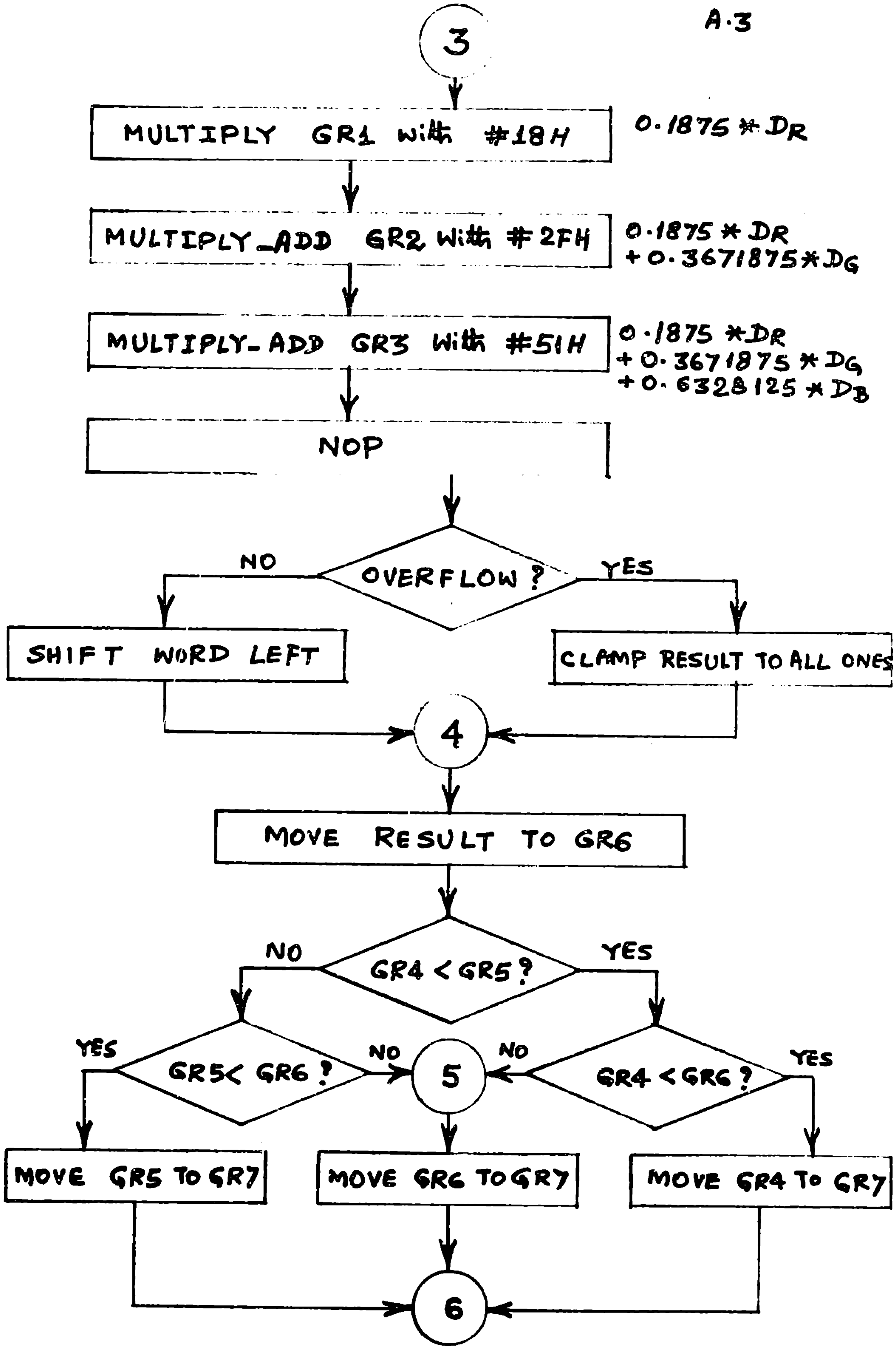


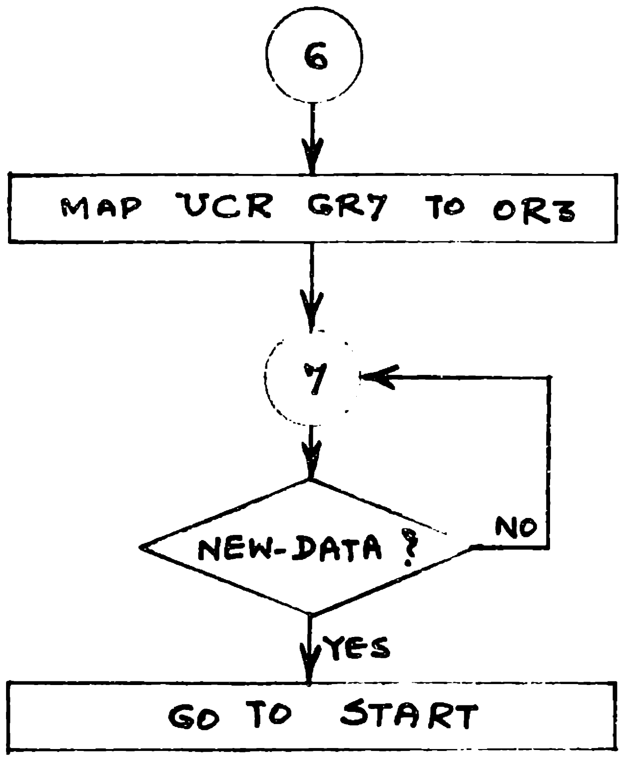

2
SPLIT FORMAT
$[\bar{Y}]_8 = \bar{Y} \wedge (37)_8$
$[\bar{C}]_8 = \bar{C} \wedge (37)_8$
$[\bar{M}]_8 = \bar{M} \wedge (37)_8$
ICT-ADDR = (3 MSB OF $\bar{Y}$) || (3 MSB OF $\bar{C}$) || (3 MSB OF $\bar{M}$)
LOOK-UP ICT ENTRIES
$T', \Delta\tilde{T}_Y, \Delta\tilde{T}_C, \Delta\tilde{T}_M, \Delta\tilde{T}_{Y,C}, \Delta\tilde{T}_{C,M}, \Delta\tilde{T}_{M,Y}$
COMPUTE
$T = T' + [\bar{Y}]_8 \Delta\tilde{T}_Y + [\bar{C}]_8 \Delta\tilde{T}_C + [\bar{M}]_8 \Delta\tilde{T}_M$
$+ [\bar{Y}]_8 [\bar{C}]_8 \Delta\tilde{T}_{Y,C}$
$+ [\bar{C}]_8 [\bar{M}]_8 \Delta\tilde{T}_{C,M}$
$+ [\bar{M}]_8 [\bar{Y}]_8 \Delta\tilde{T}_{M,Y}$
OUTPUT T
T = one of {Y, C, M}
3
NEW-DATA?
NO
YES
GO TO 1

MICRO-FLOWCHART FOR K

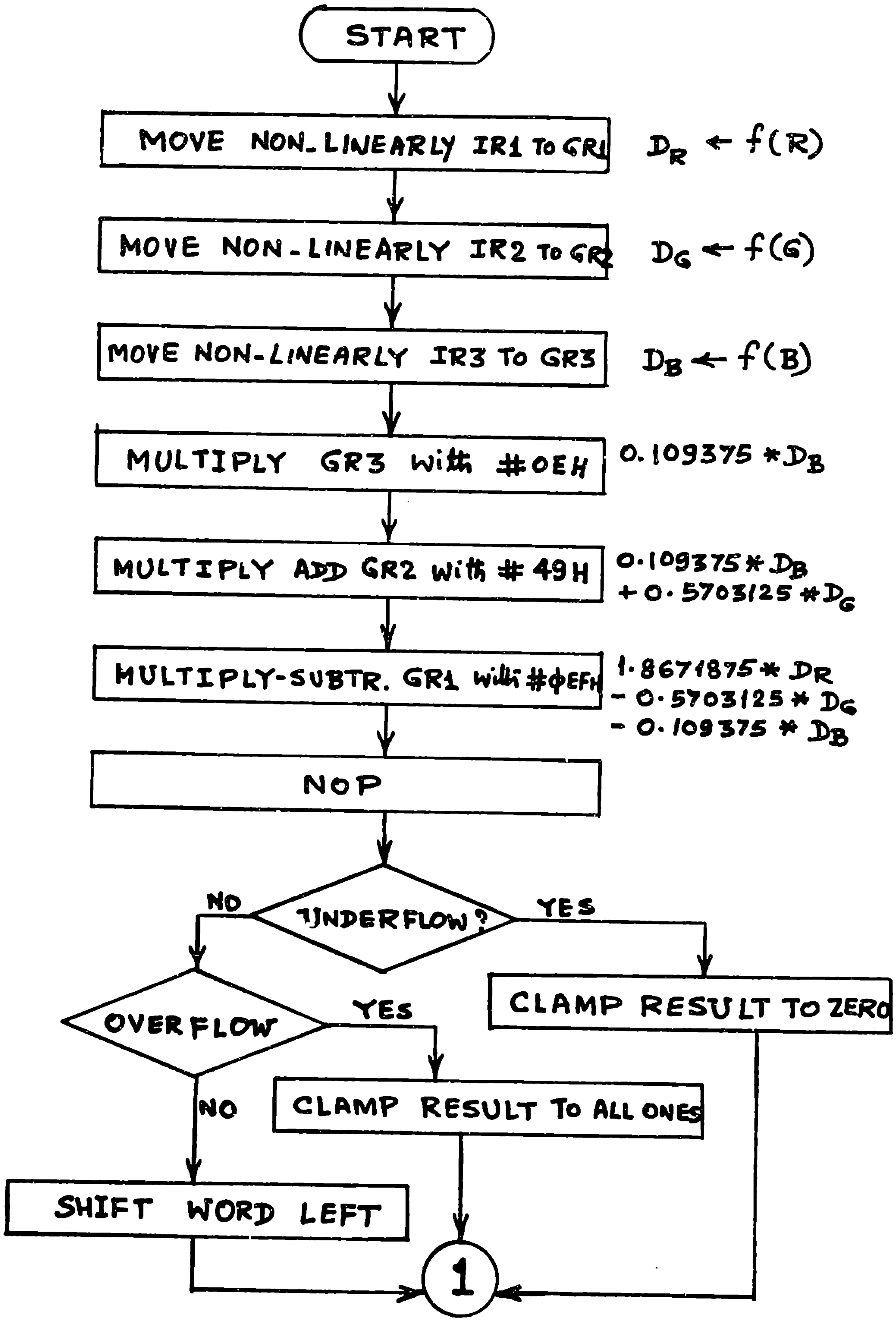



1
MOVE RESULT TO GR4
MULTIPLY GR3 with #23H
0.2734375 * D_B
MULTIPLY-ADD GR1 with #94H
0.2734375 * D_B
+ 1.15625 * D_R
MULTIPLY-SUBTR. GR2 with #ØFFH
1.9921875 * D_G
- 1.15625 * D_R
- 0.2734375 * D_B
NOP
UNDERFLOW?
NO
YES
CLAMP RESULT TO ZERO
OVERFLOW?
YES
NO
CLAMP RESULT TO ALL ONES
SHIFT WORD LEFT
2
MOVE RESULT TO GR5
3

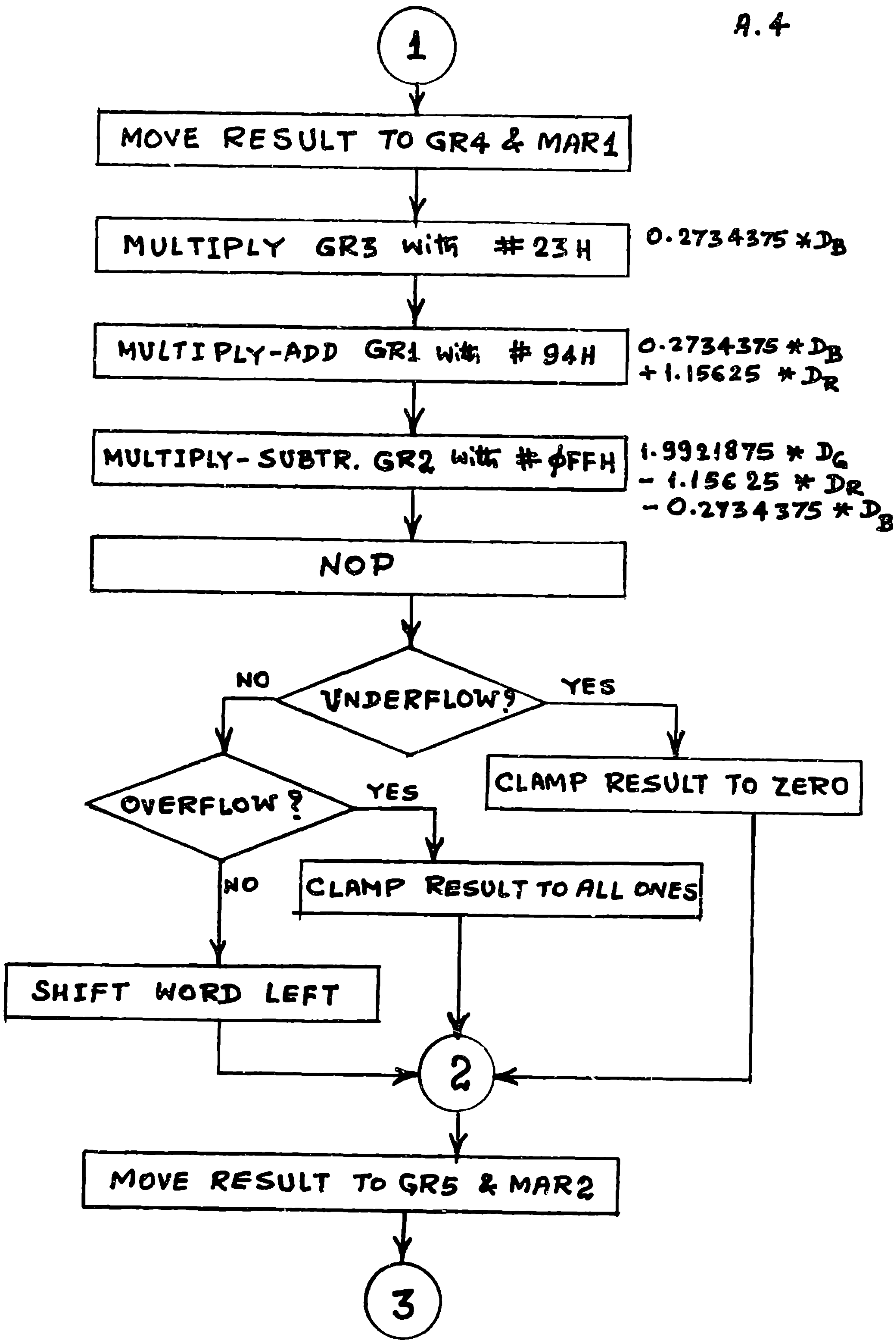

3
MULTIPLY GR1 with #18H
0.1875 * DR
MULTIPLY_ADD GR2 with #2FH
0.1875 * DR
+ 0.3671875 * DG
MULTIPLY-ADD GR3 with #51H
0.1875 * DR
+ 0.3671875 * DG
+ 0.6328125 * DB
NOP
OVERFLOW ?
NO
YES
SHIFT WORD LEFT
CLAMP RESULT TO ALL ONES
4
MOVE RESULT TO GR6
GR4 < GR5 ?
NO
YES
GR5 < GR6 ?
YES
NO
5
NO
GR4 < GR6 ?
YES
MOVE GR5 TO GR7
MOVE GR6 TO GR7
MOVE GR4 TO GR7
6

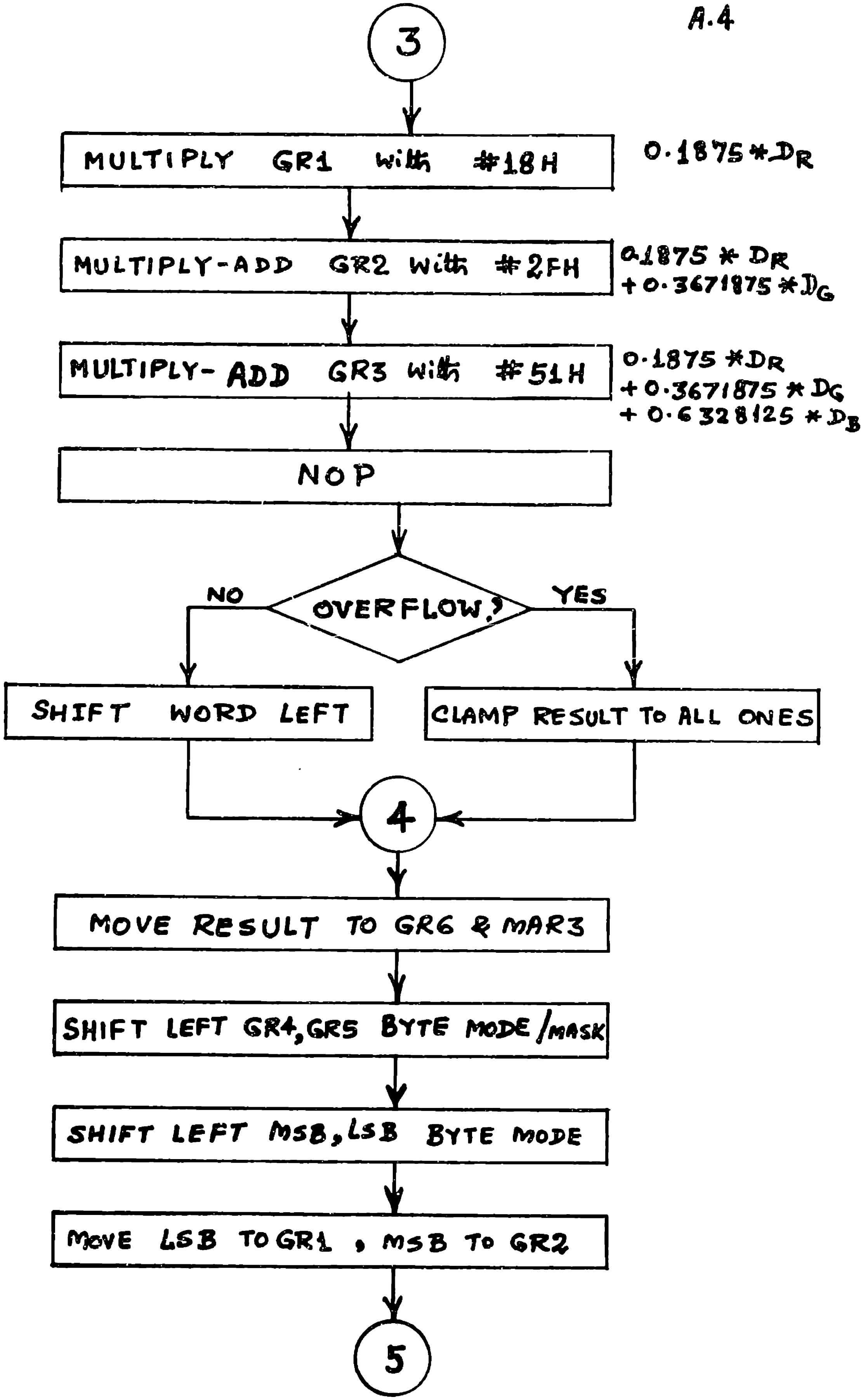

6
MAP UCR GR7 TO OR3
7
NEW-DATA ?
NO
YES
GO TO START

MICRO-FLOW CHART FOR Y, C, M

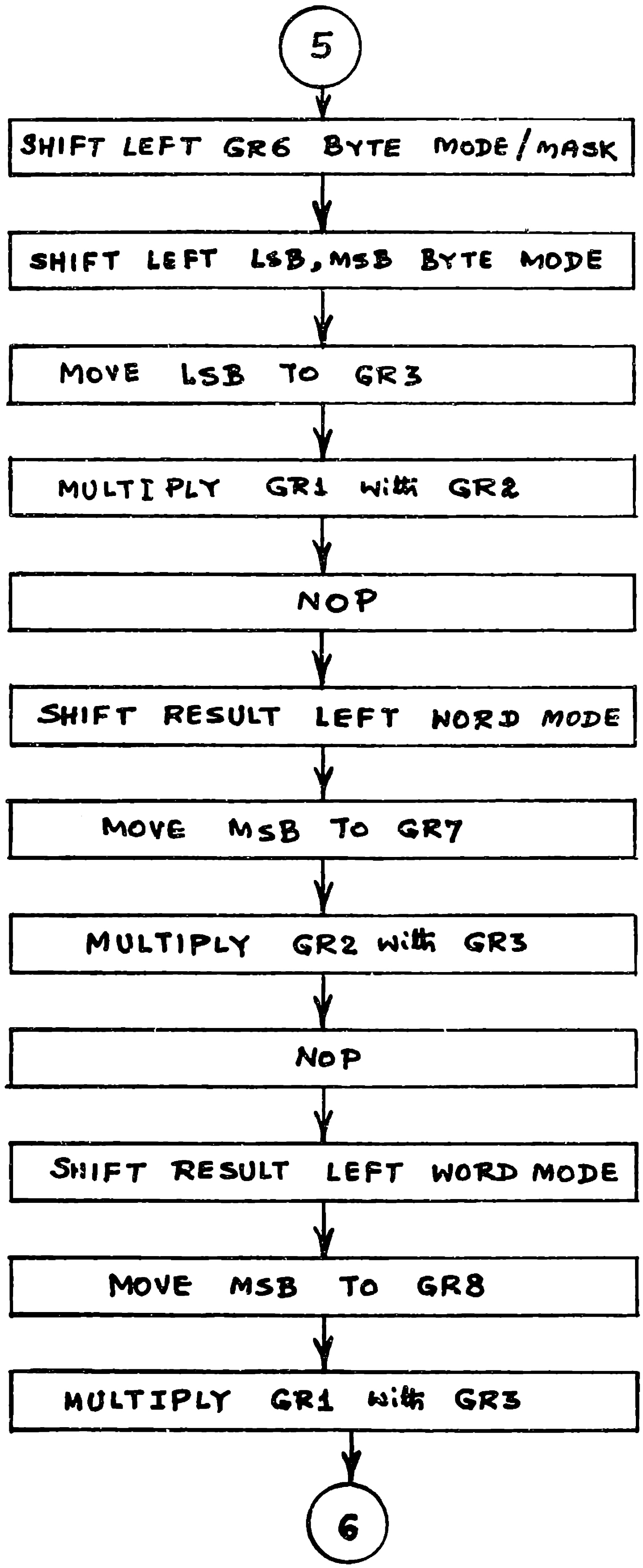


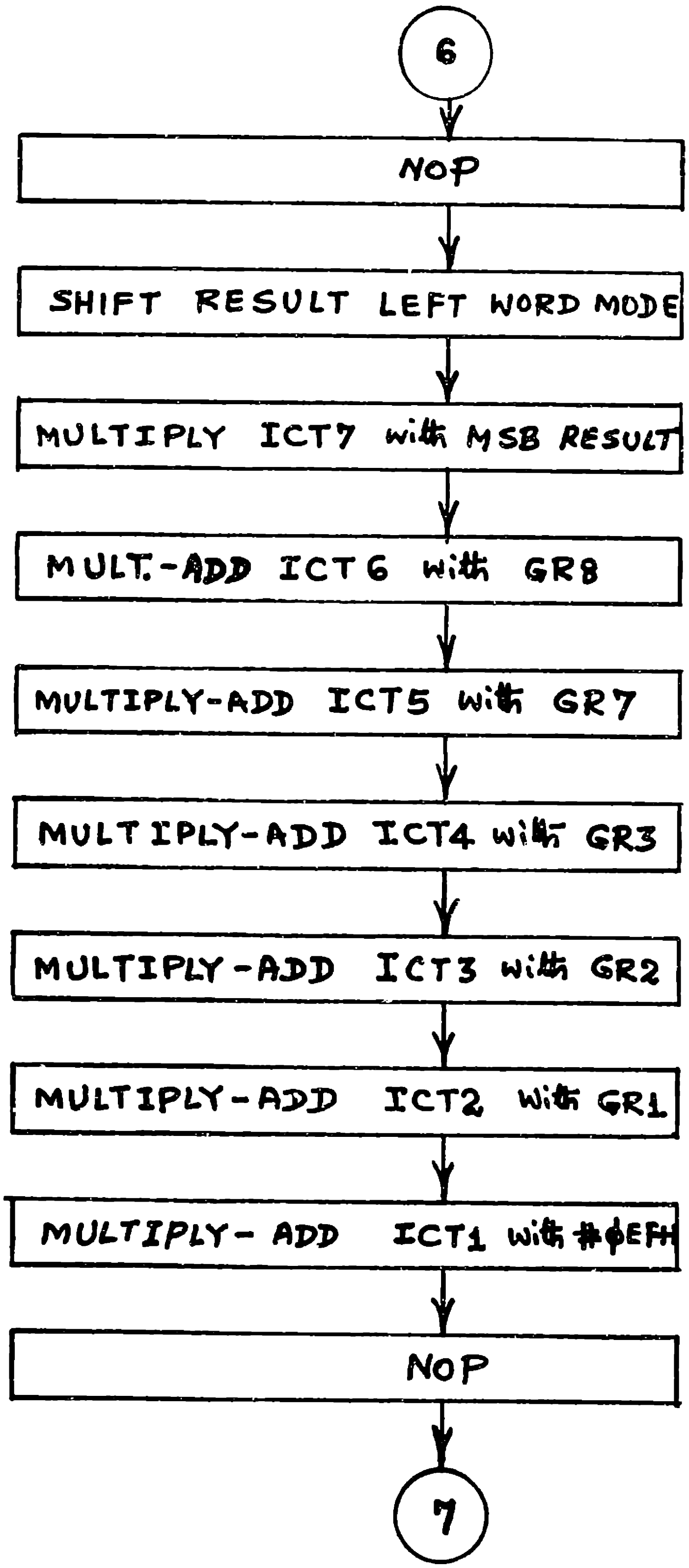
A.4

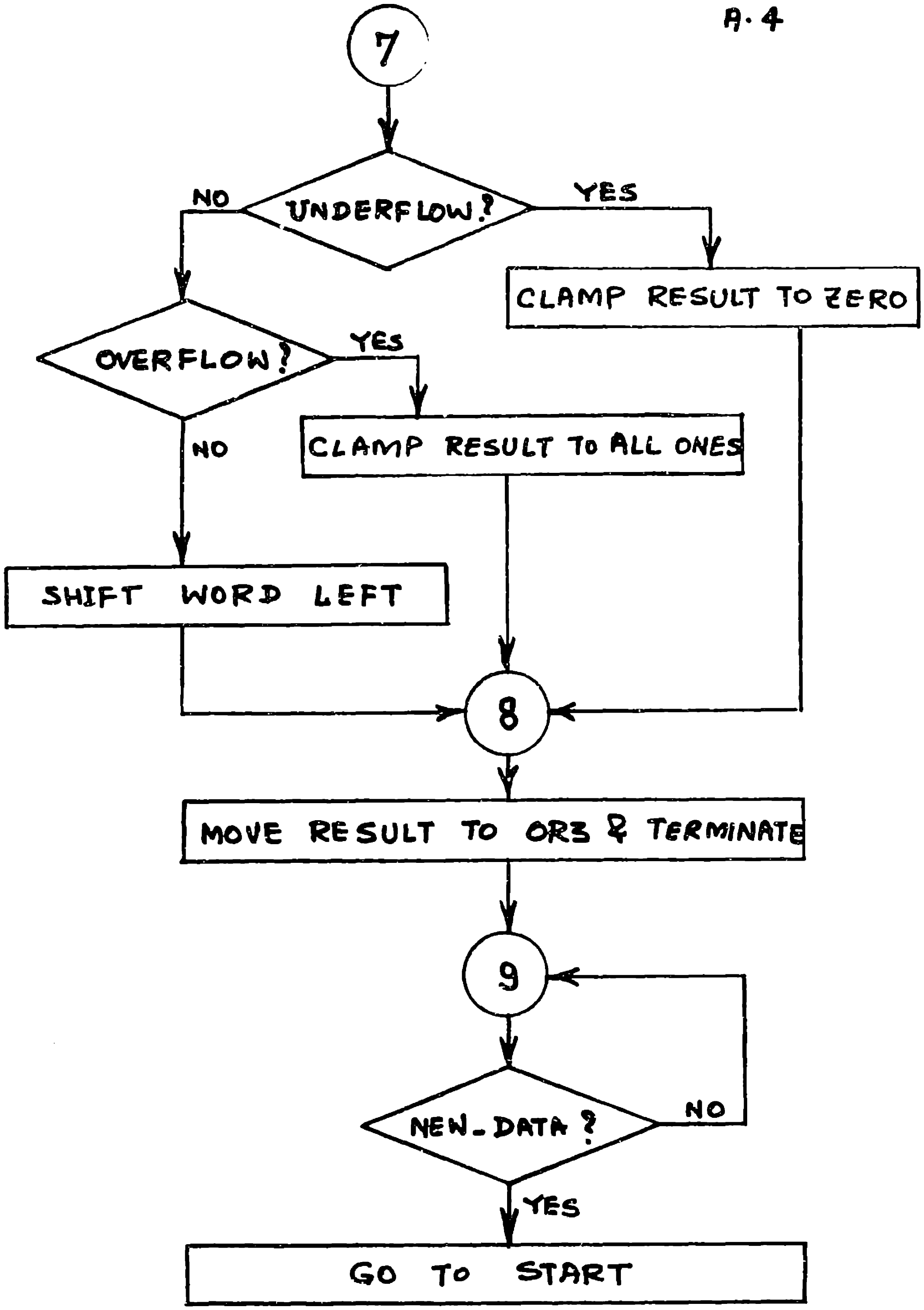

1
MOVE RESULT TO GR4 & MAR1
MULTIPLY GR3 with #23H
0.2734375 * DB
MULTIPLY-ADD GR1 with #94H
0.2734375 * DB
+ 1.15625 * DR
MULTIPLY-SUBTR. GR2 with #ØFFH
1.9921875 * DG
- 1.15625 * DR
- 0.2734375 * DB
NOP
UNDERFLOW?
NO
YES
CLAMP RESULT TO ZERO
OVERFLOW?
YES
NO
CLAMP RESULT TO ALL ONES
SHIFT WORD LEFT
2
MOVE RESULT TO GR5 & MAR2
3



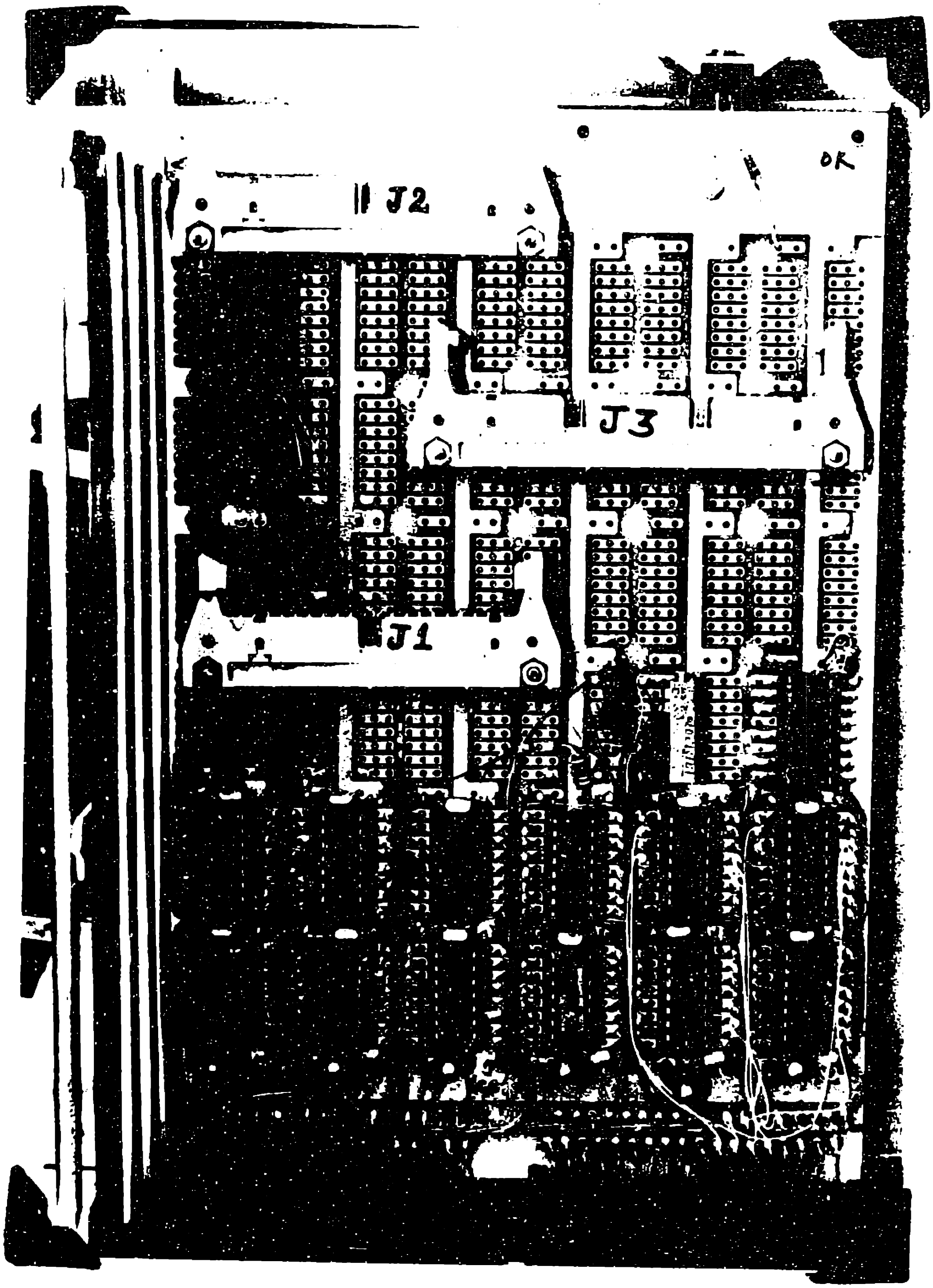
3
MULTIPLY GR1 with #18H
0.1875*DR
MULTIPLY-ADD GR2 with #2FH
0.1875 * DR
+0.3671875 *DG
MULTIPLY-ADD GR3 with #51H
0.1875 *DR
+0.3671875 * DG
+0.6328125 * DB
NOP
OVERFLOW?
NO
YES
SHIFT WORD LEFT
CLAMP RESULT TO ALL ONES
4
MOVE RESULT TO GR6 & MAR3
SHIFT LEFT GR4, GR5 BYTE MODE/MASK
SHIFT LEFT MSB, LSB BYTE MODE
MOVE LSB TO GR1, MSB TO GR2
5


5
SHIFT LEFT GR6 BYTE MODE/MASK
SHIFT LEFT LSB,MSB BYTE MODE
MOVE LSB TO GR3
MULTIPLY GR1 with GR2
NOP
SHIFT RESULT LEFT WORD MODE
MOVE MSB TO GR7
MULTIPLY GR2 with GR3
NOP
SHIFT RESULT LEFT WORD MODE
MOVE MSB TO GR8
MULTIPLY GR1 with GR3
6

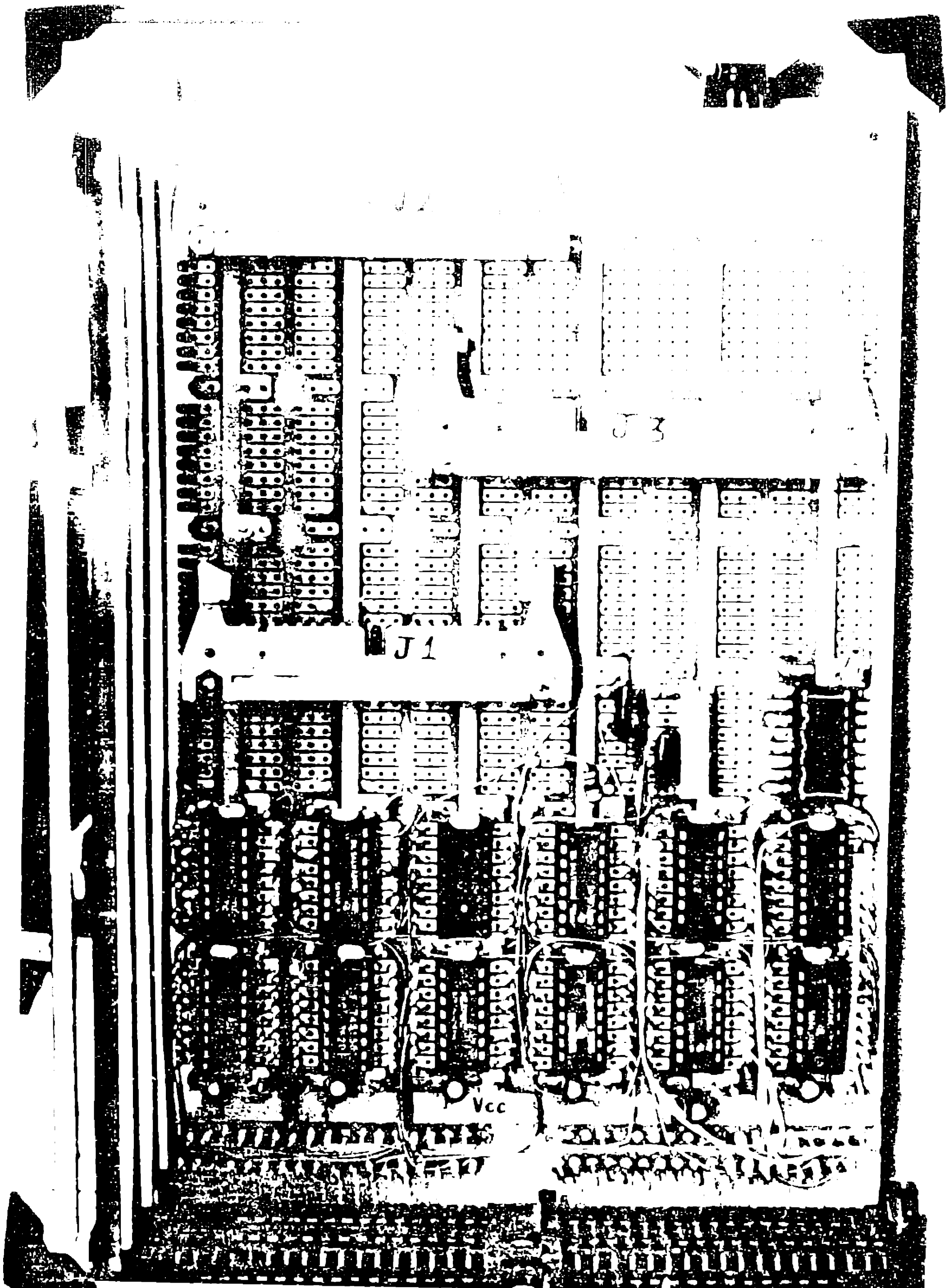

6
NOP
SHIFT RESULT LEFT WORD MODE
MULTIPLY ICT7 with MSB RESULT
MULT.-ADD ICT6 with GR8
MULTIPLY-ADD ICT5 with GR7
MULTIPLY-ADD ICT4 with GR3
MULTIPLY-ADD ICT3 with GR2
MULTIPLY-ADD ICT2 with GR1
MULTIPLY- ADD ICT1 with #ϕEFH
NOP
7


7
UNDERFLOW?
NO
YES
CLAMP RESULT TO ZERO
OVERFLOW?
YES
NO
CLAMP RESULT TO ALL ONES
SHIFT WORD LEFT
8
MOVE RESULT TO OR3 & TERMINATE
9
NEW_DATA ?
NO
YES
GO TO START

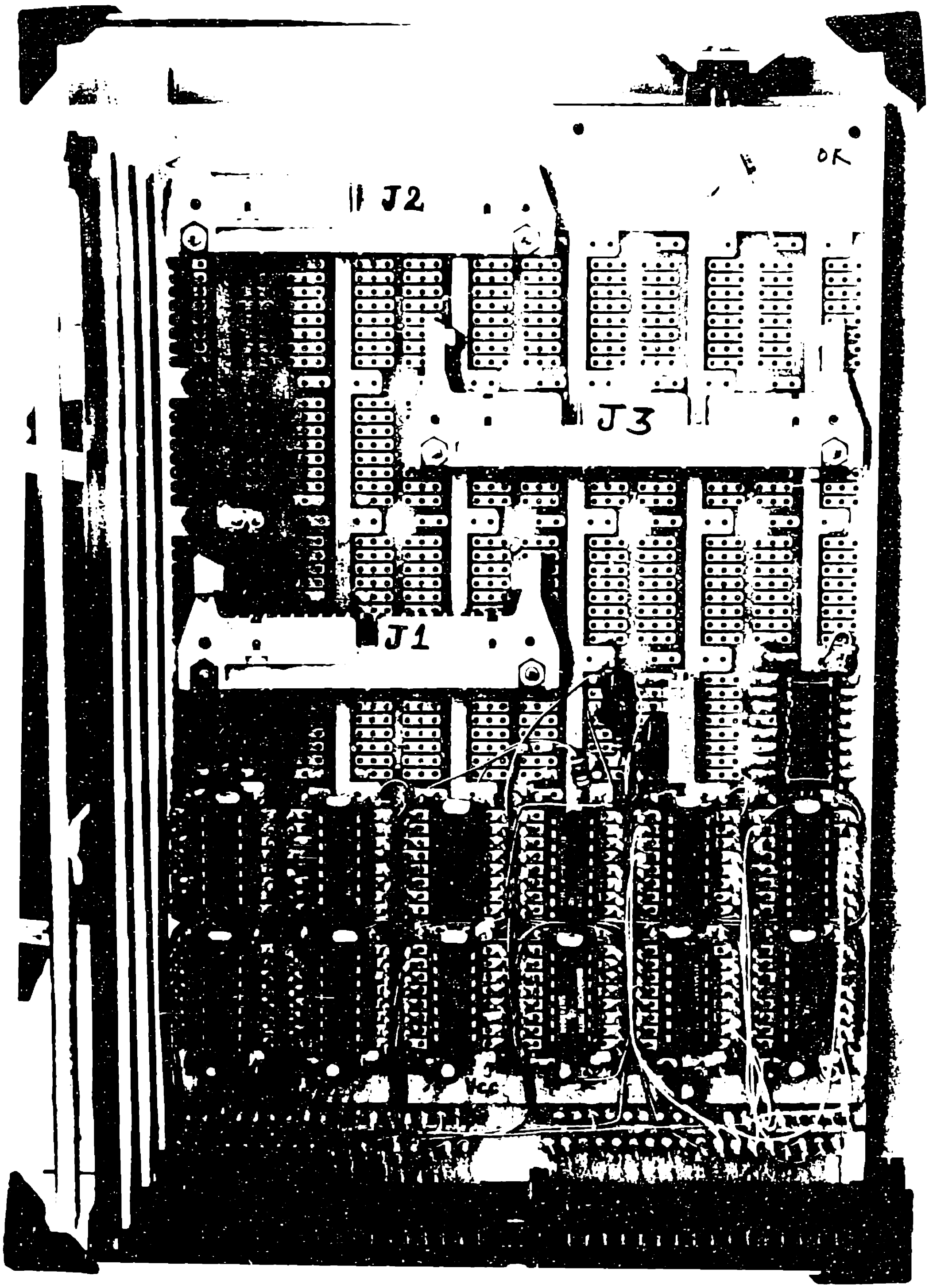
APPENDIX-B



DATA INTERFACE

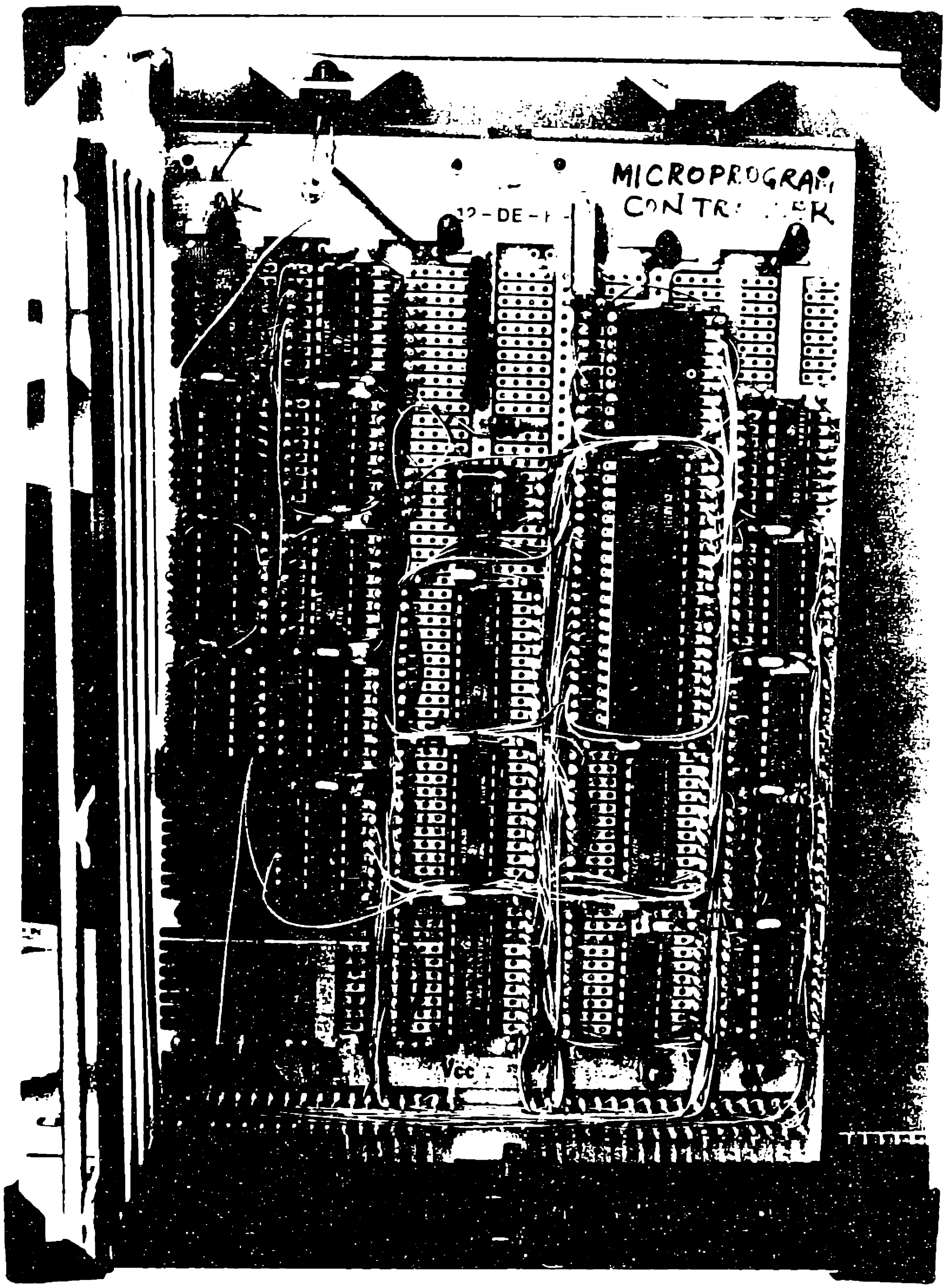

J3
J1
Vcc

APPENDIX-B



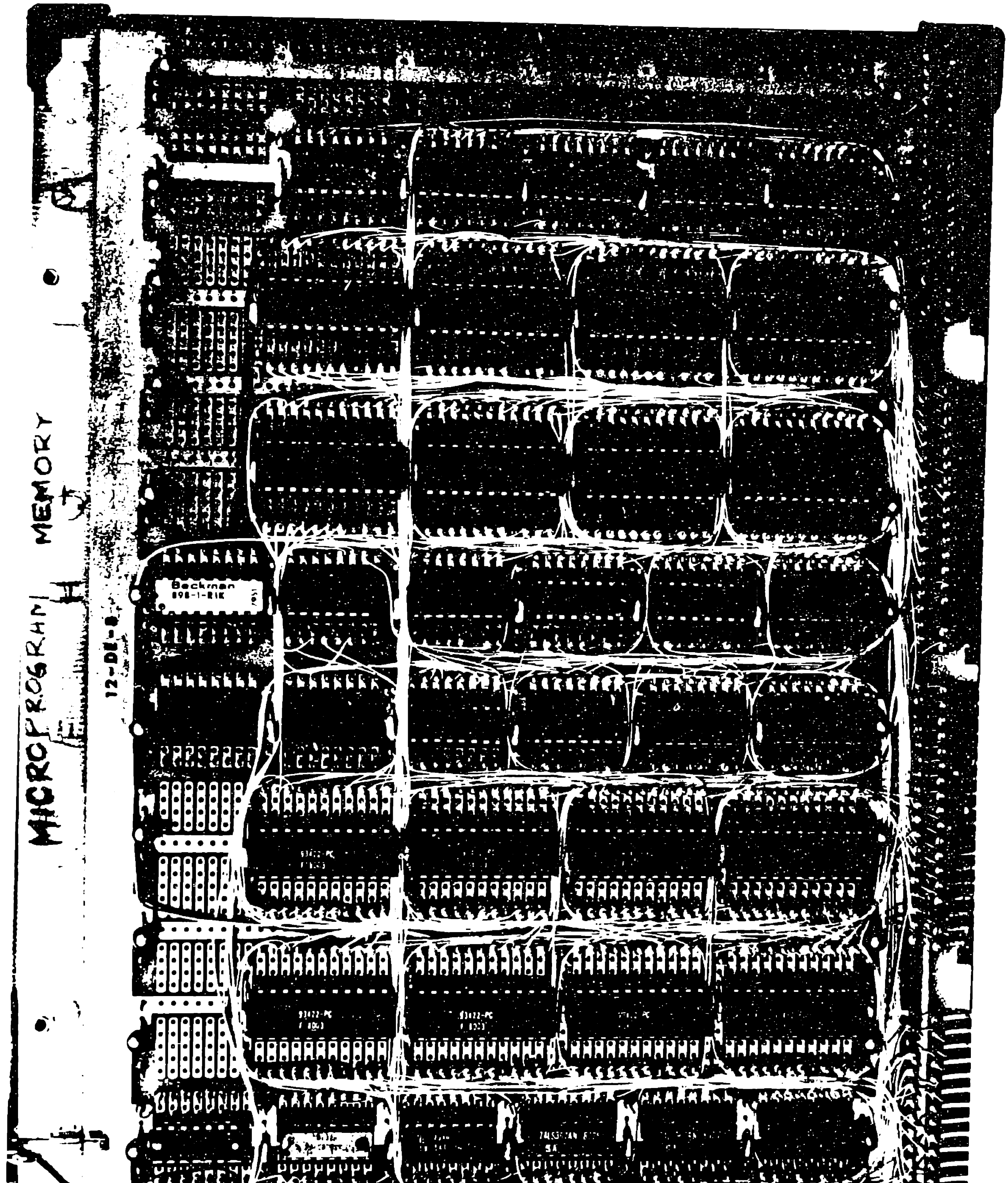
DATA INTERFACE

app - B

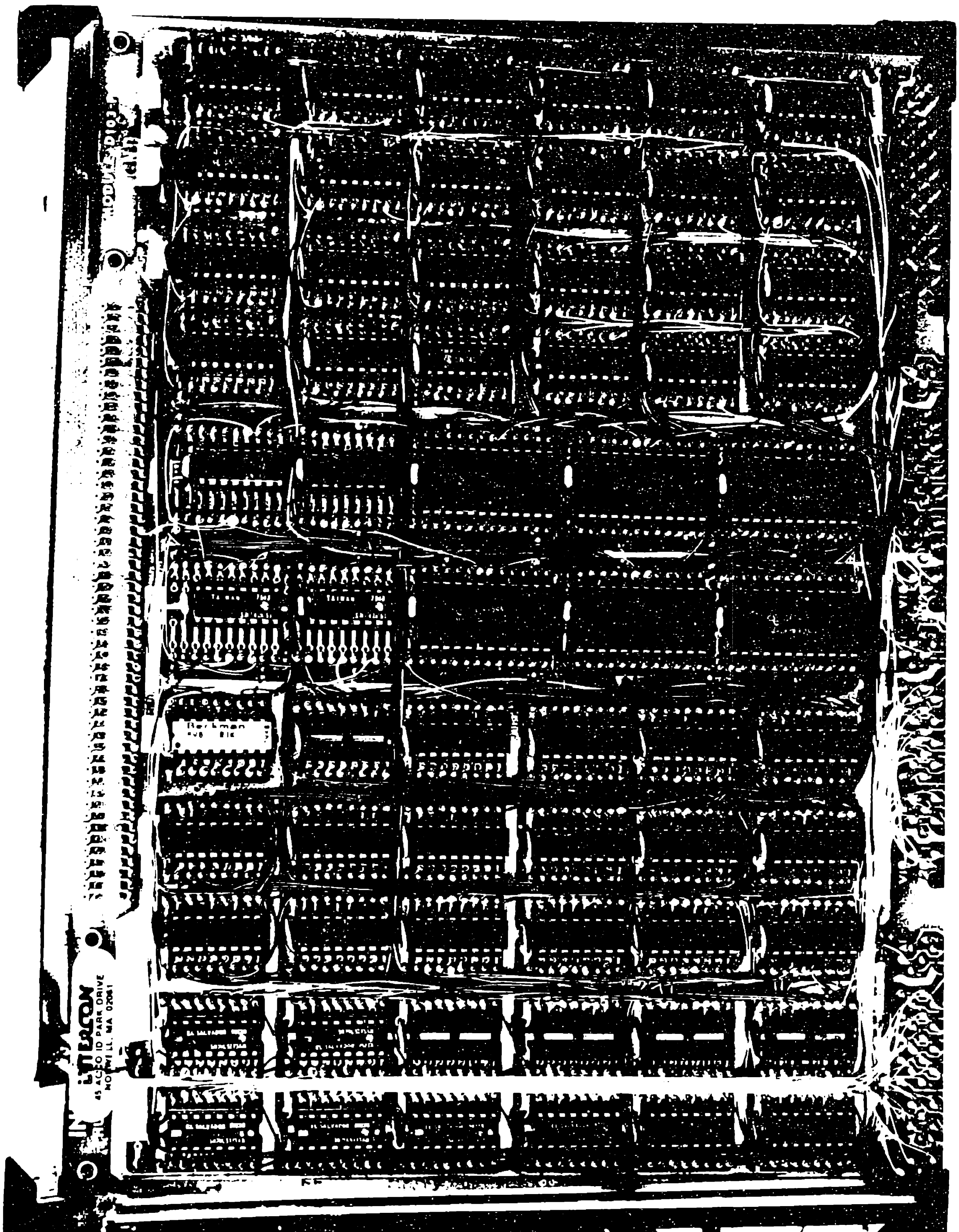


MICROPROGRAM CONTROLLER

app-B

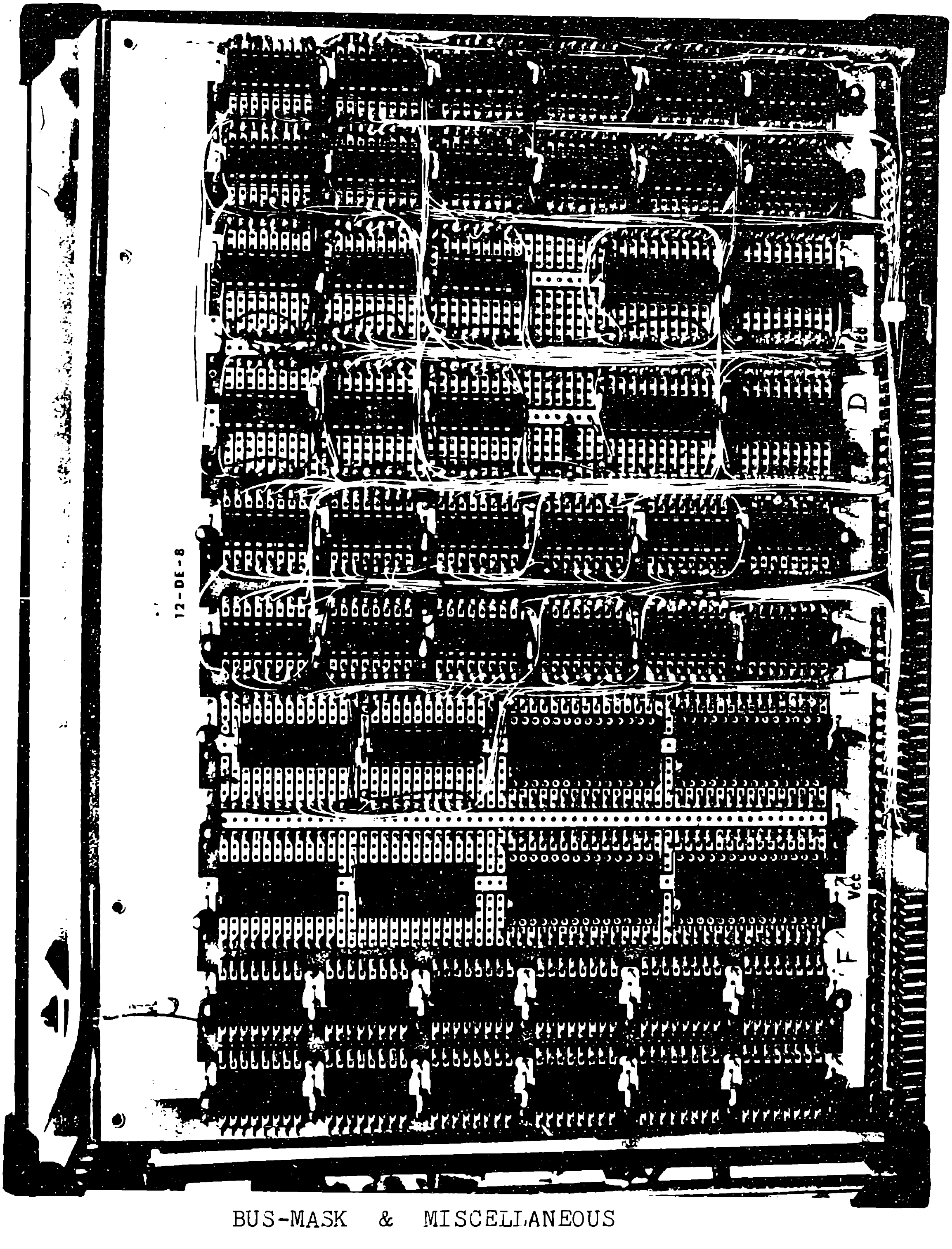


MICROPROGRAM MEMORY

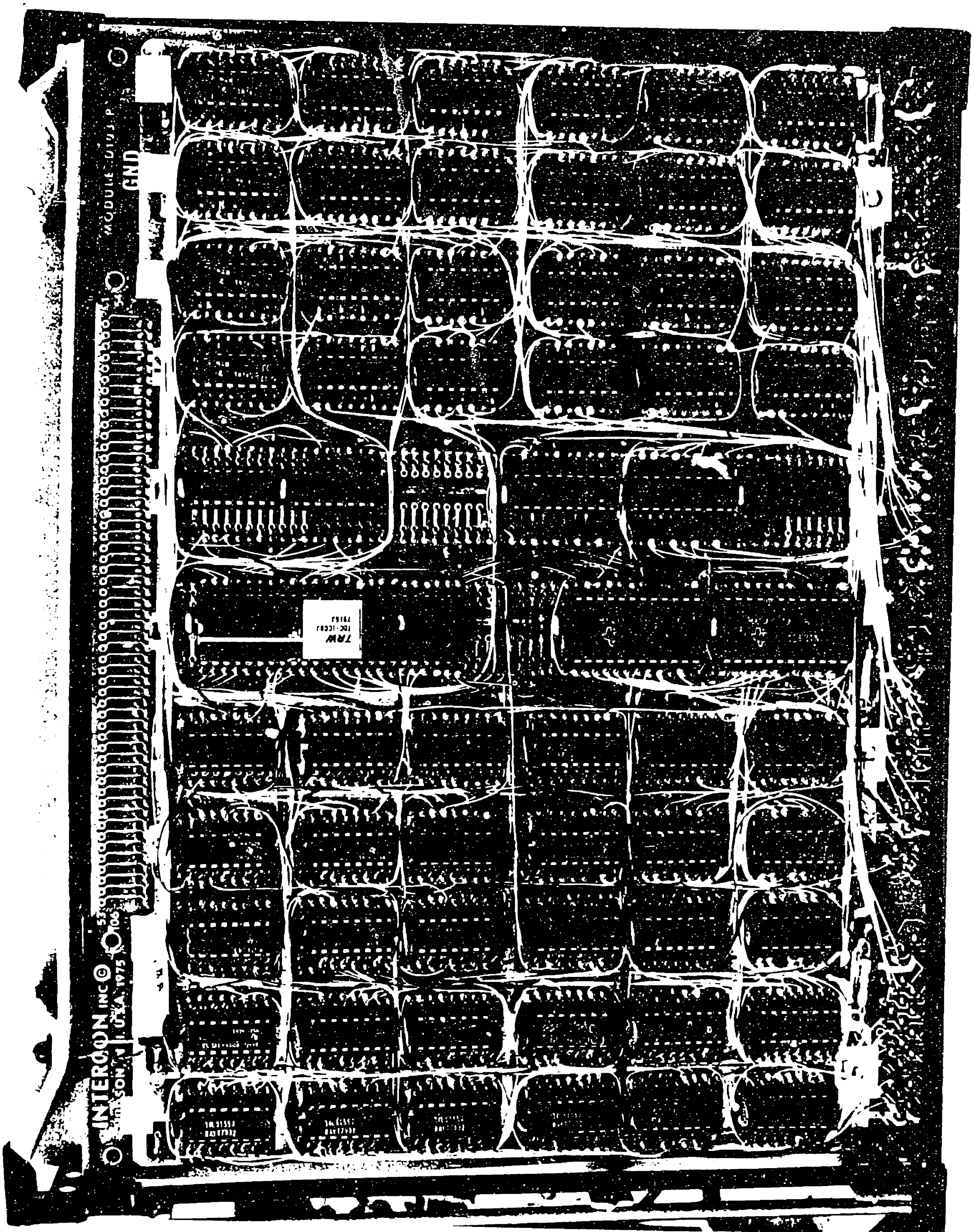


REGISTER ARRAY & BUS-CONTROL LOGIC

app - B

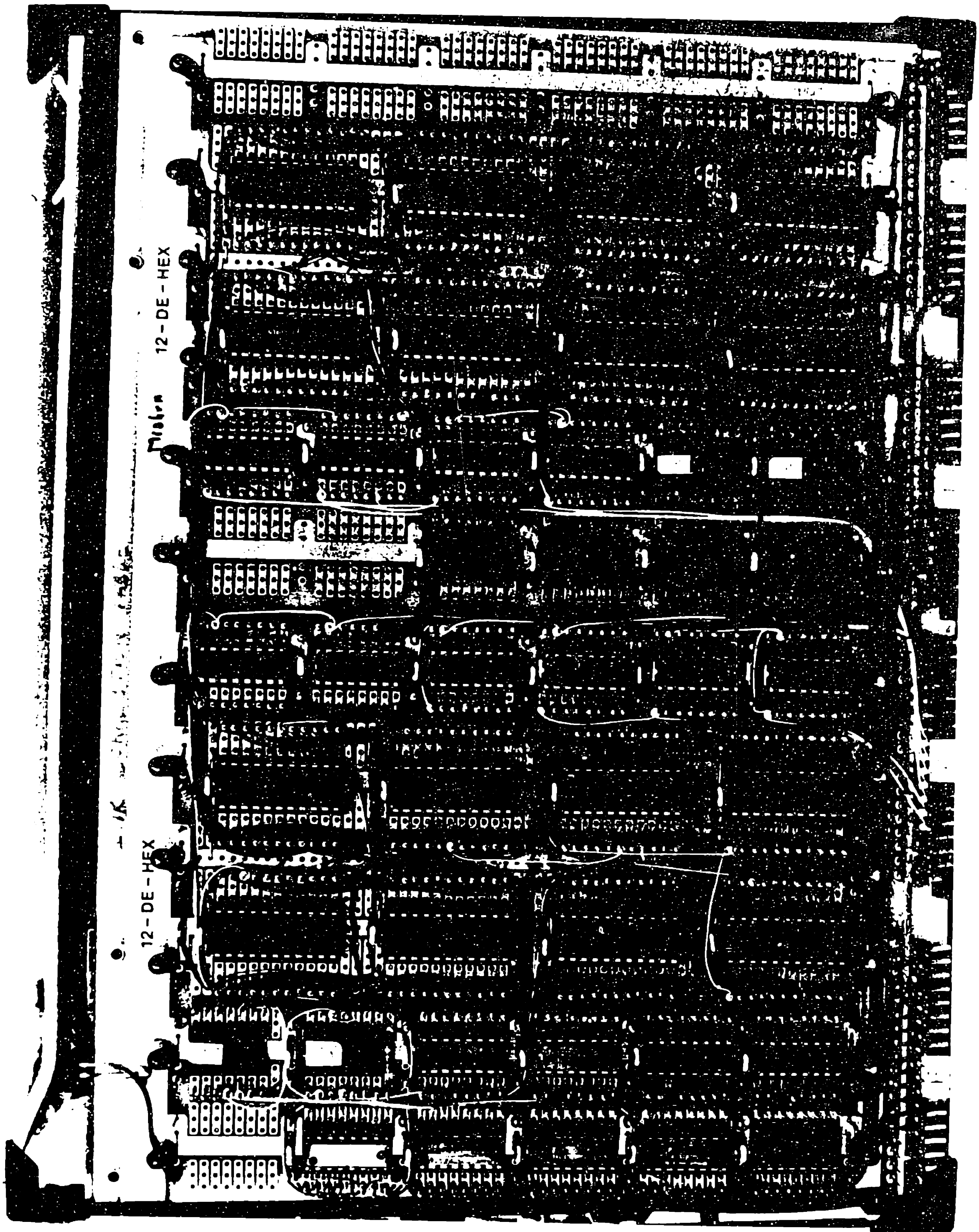


BUS-MASK & MISCELLANEOUS



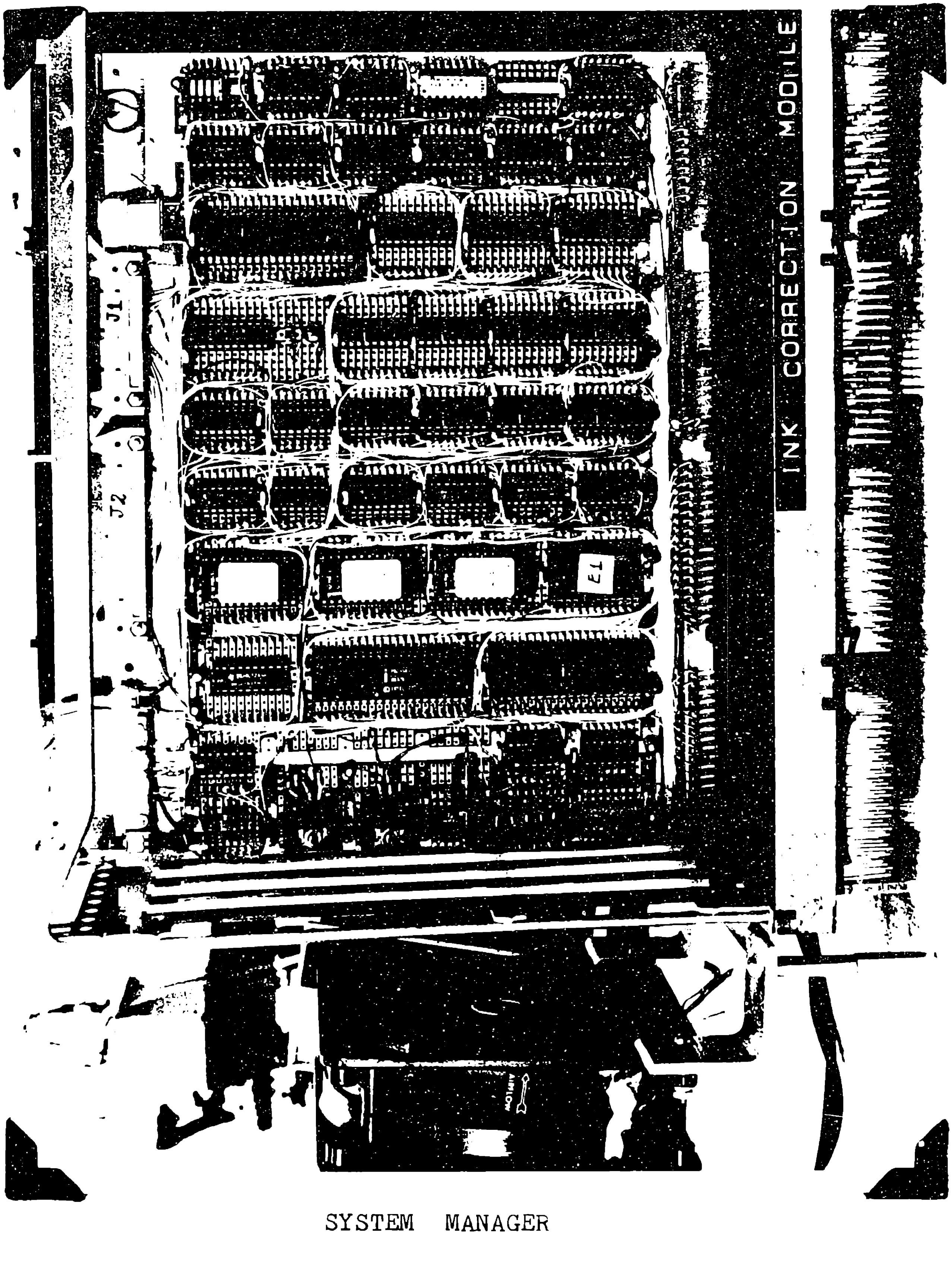
ARITHMETIC & LOGIC EXECUTIVE

app B



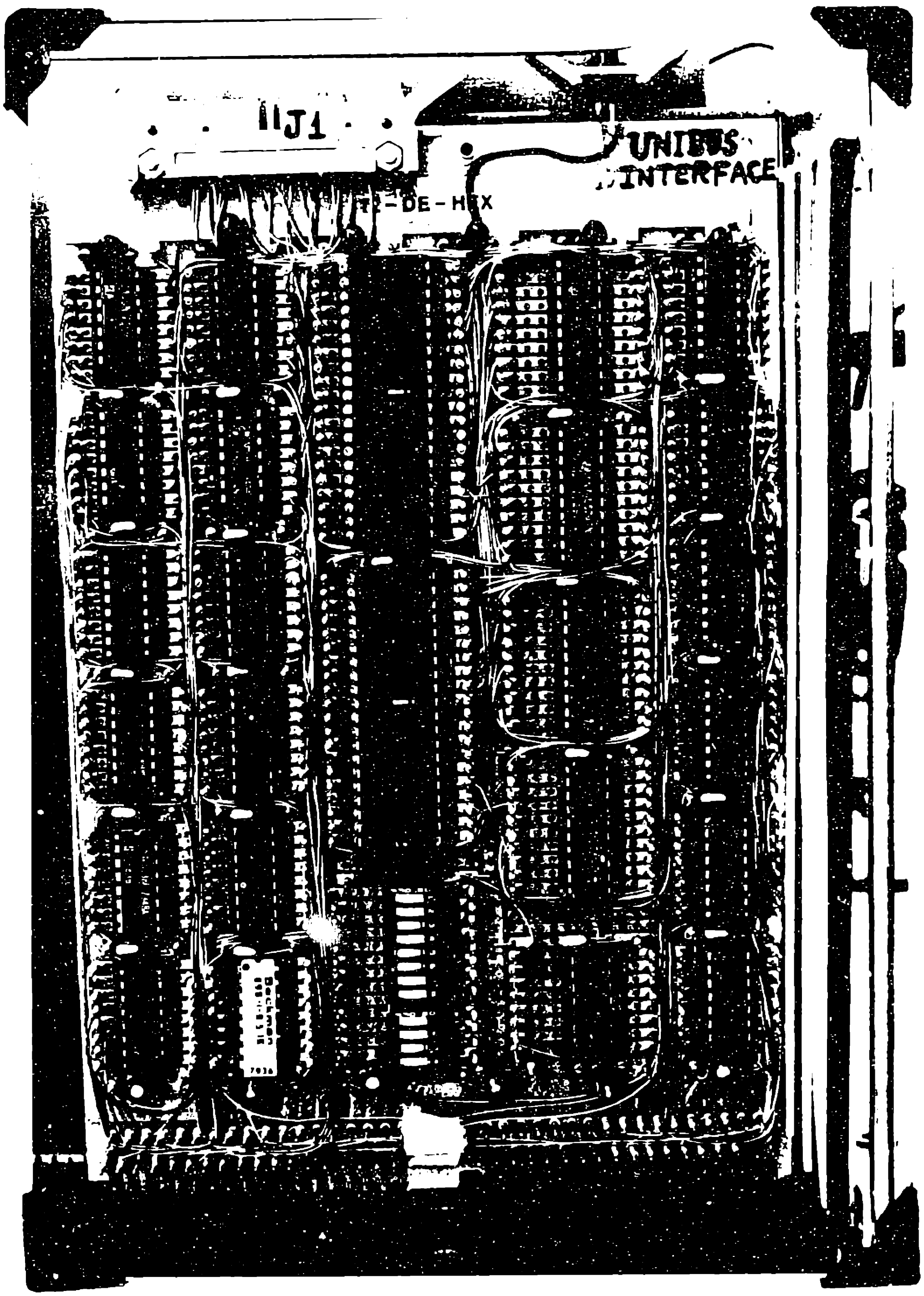
INK CORRECTION TABLE

app B



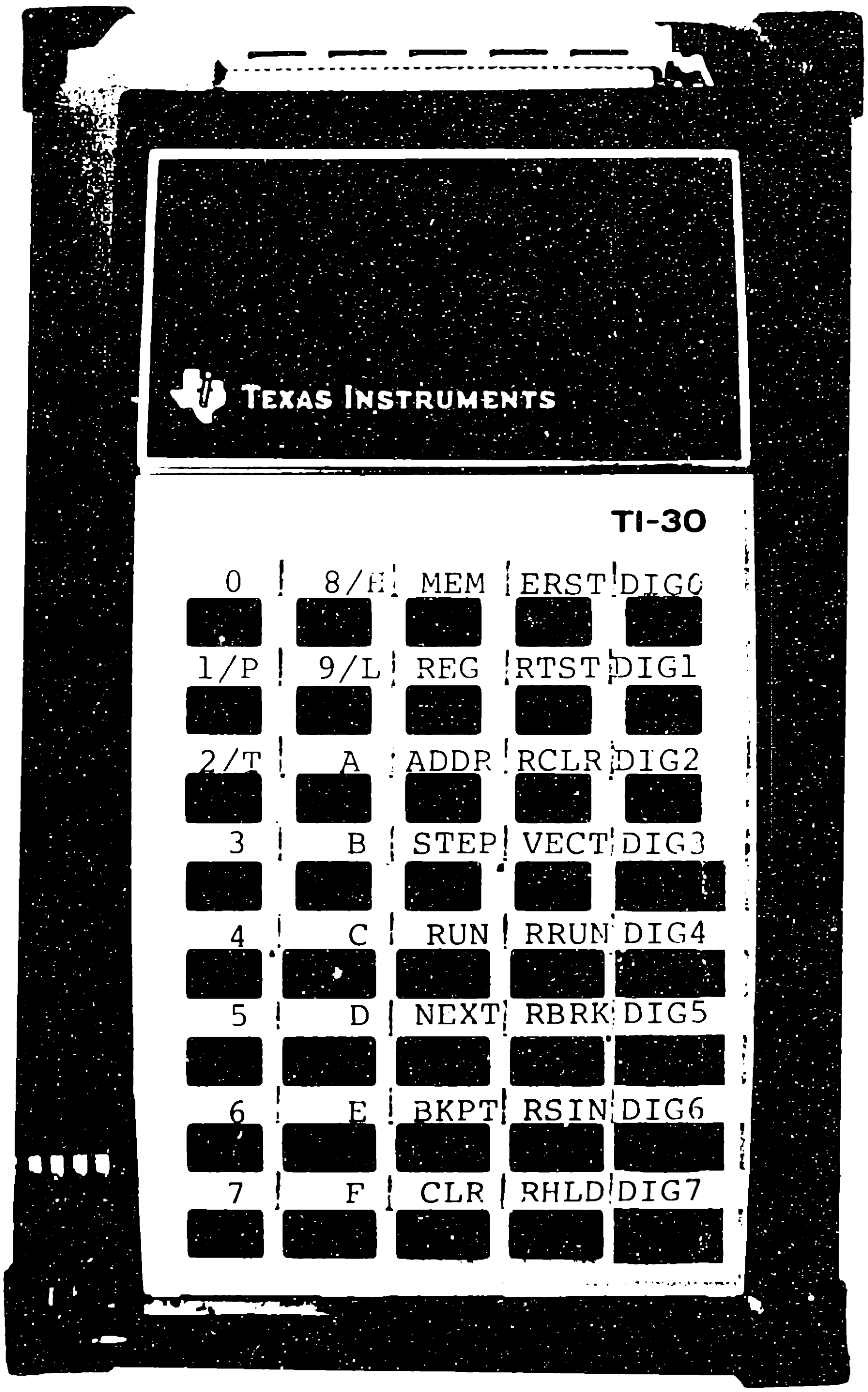
SYSTEM MANAGER

app B



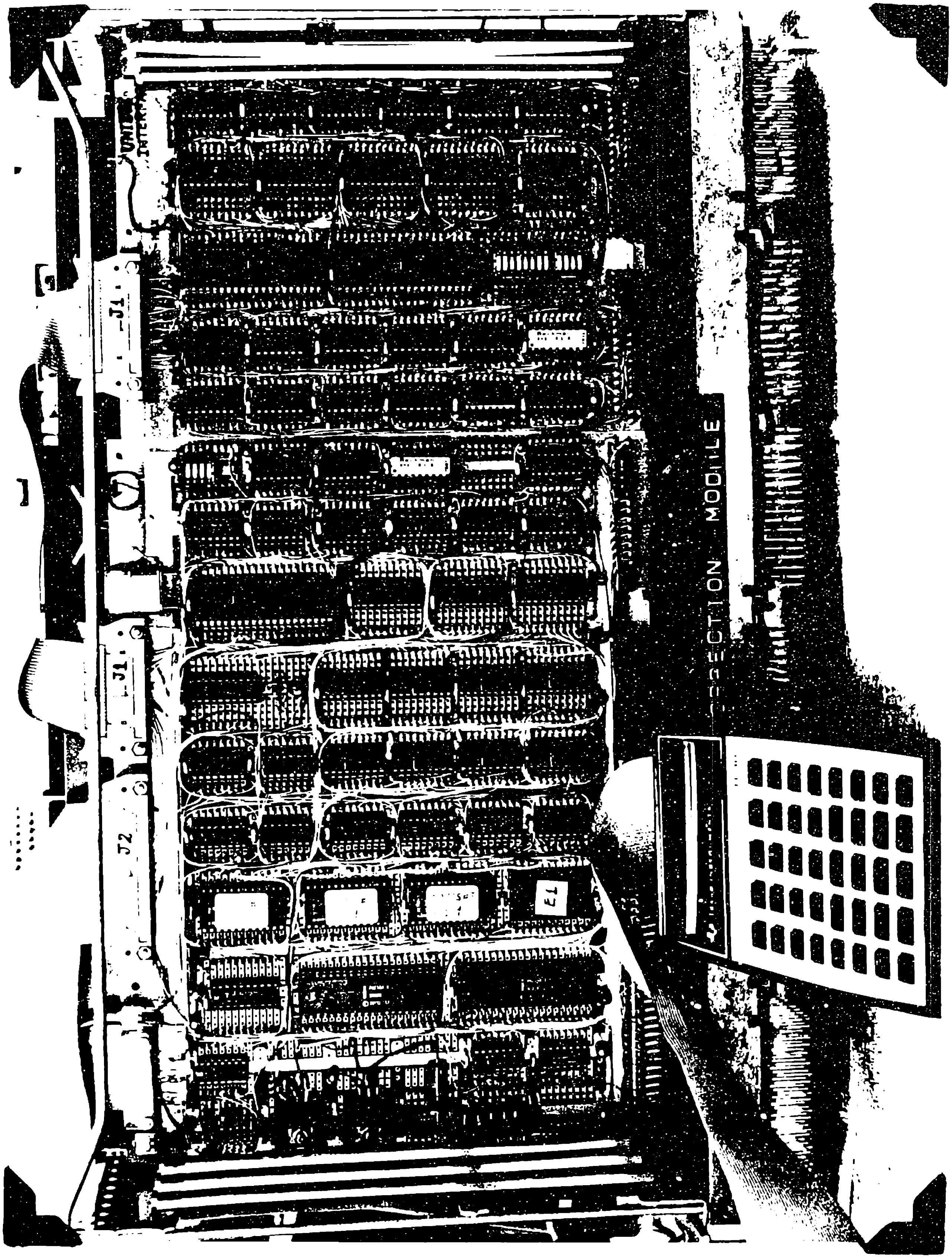
UNIBUS INTERFACE

app B



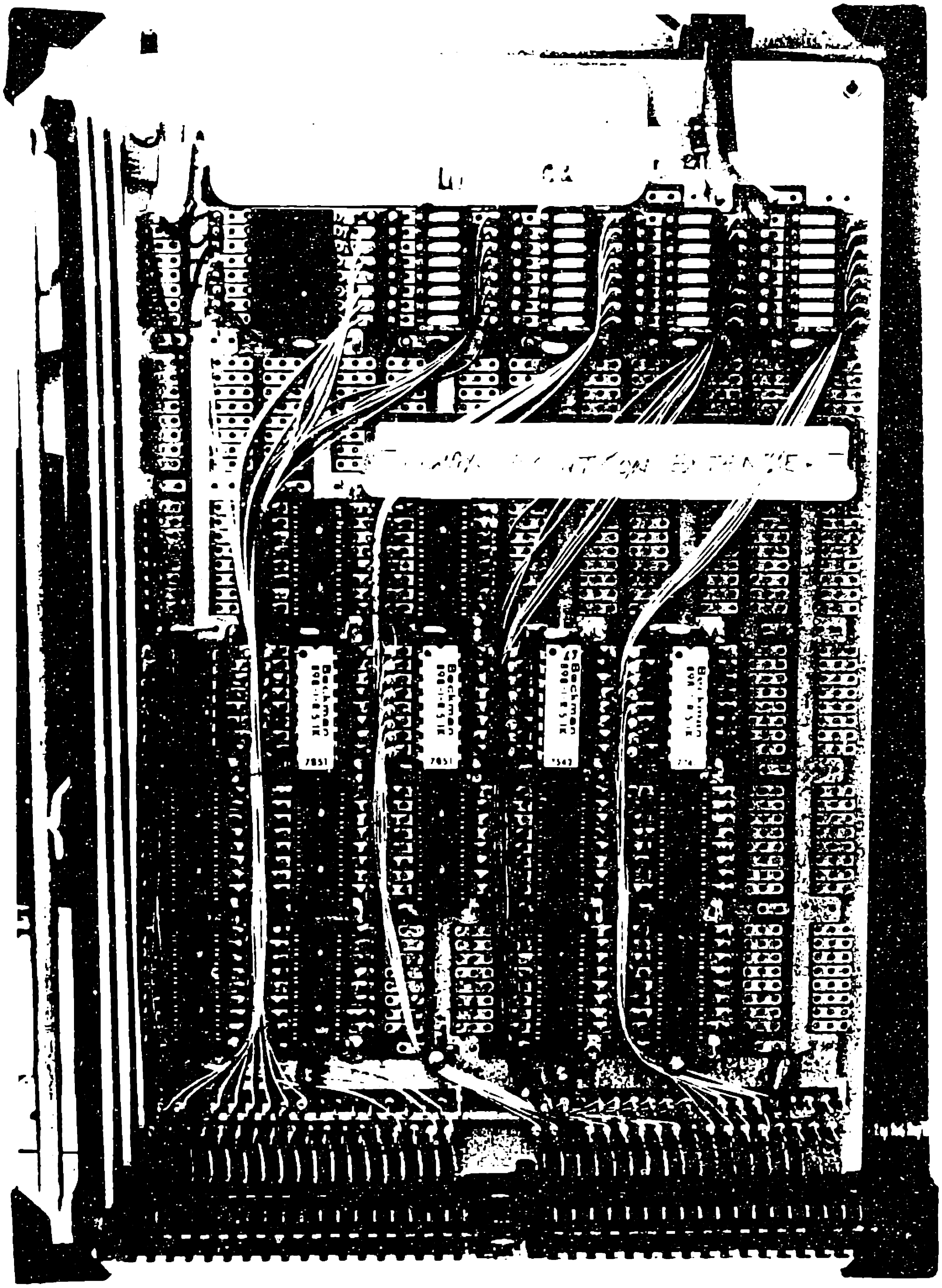
HHKD MODULE

app-B



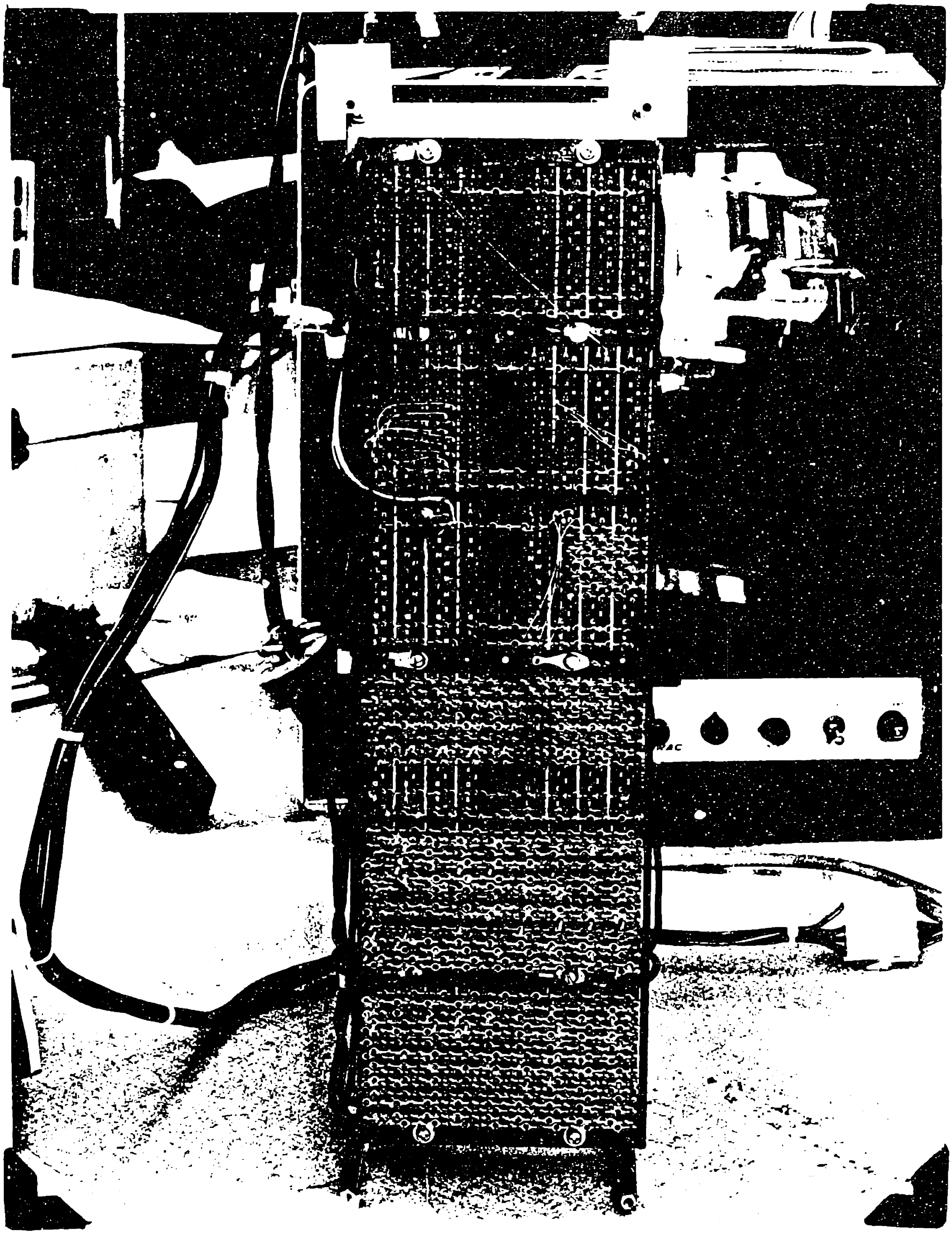
SYSTEM MANAGER WITH INTERFACES

app-B



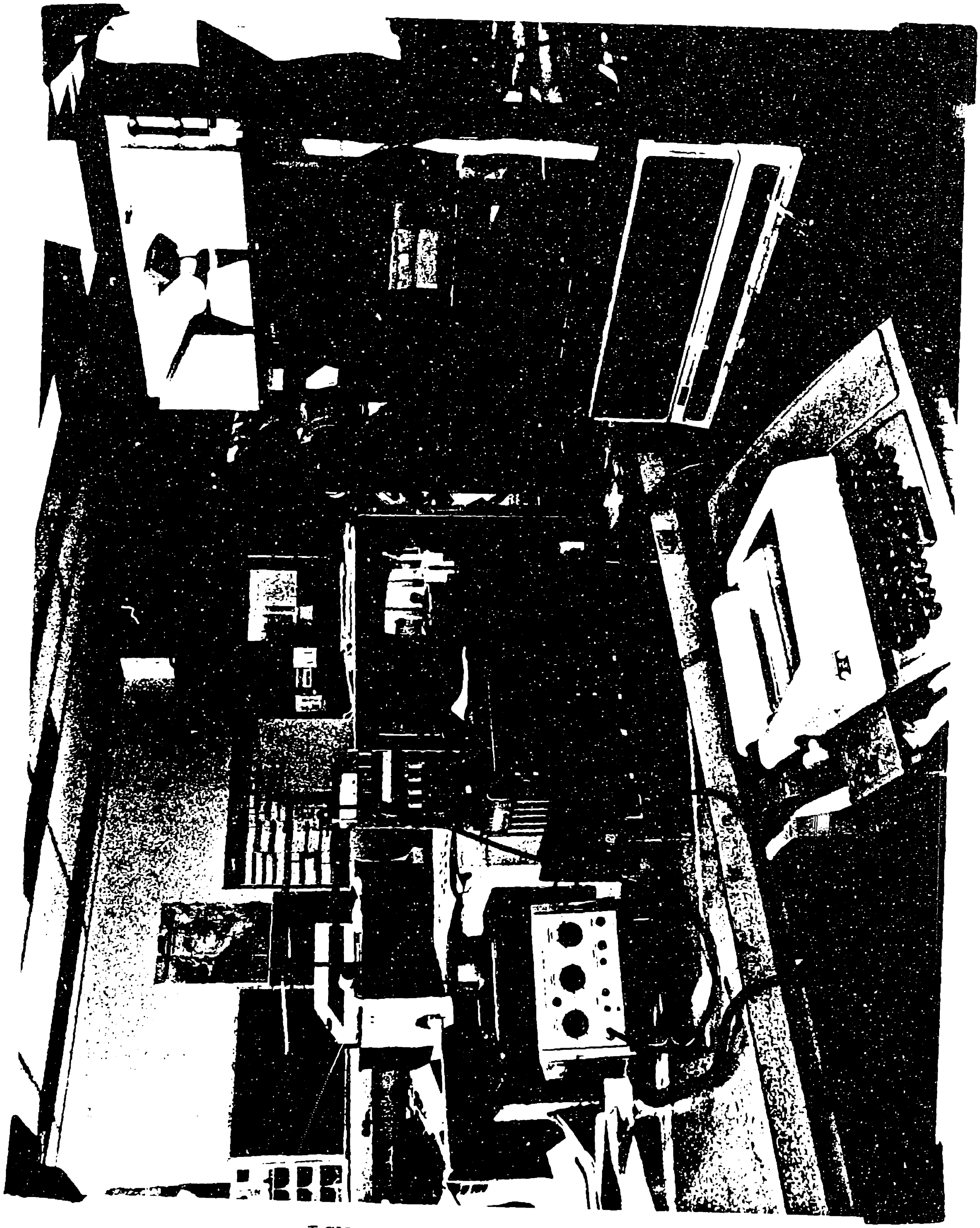
ICM TEST BOARD

app-B



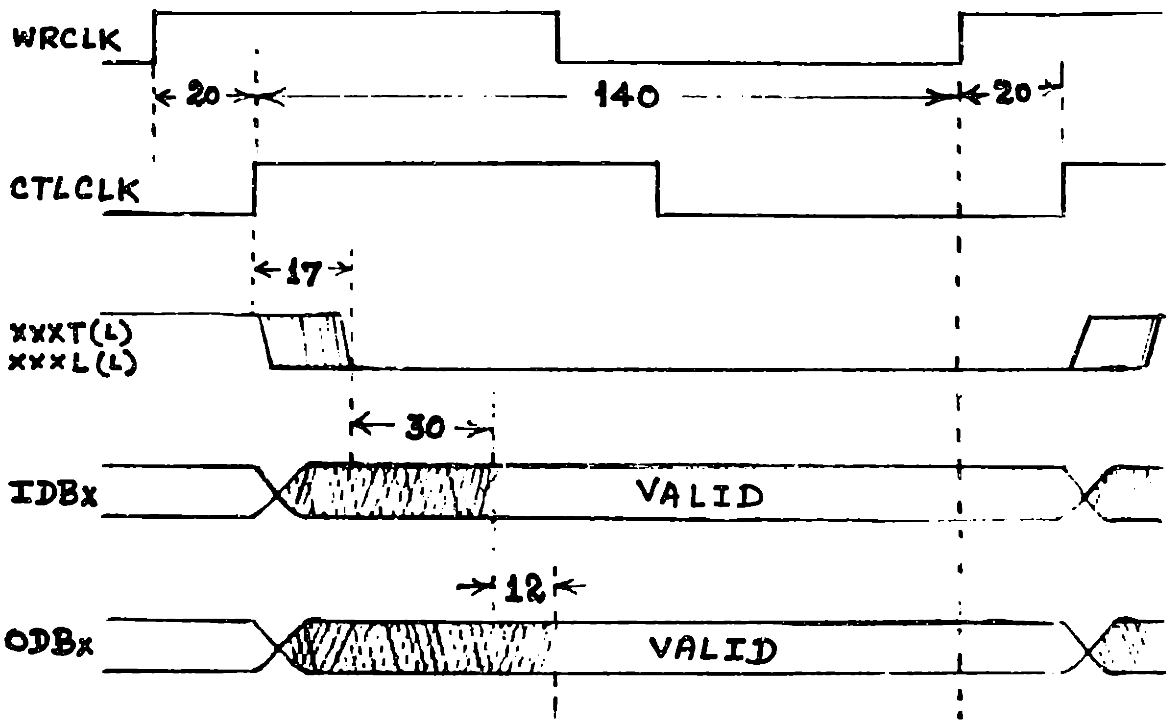
ICM BACK PLANE

app-B

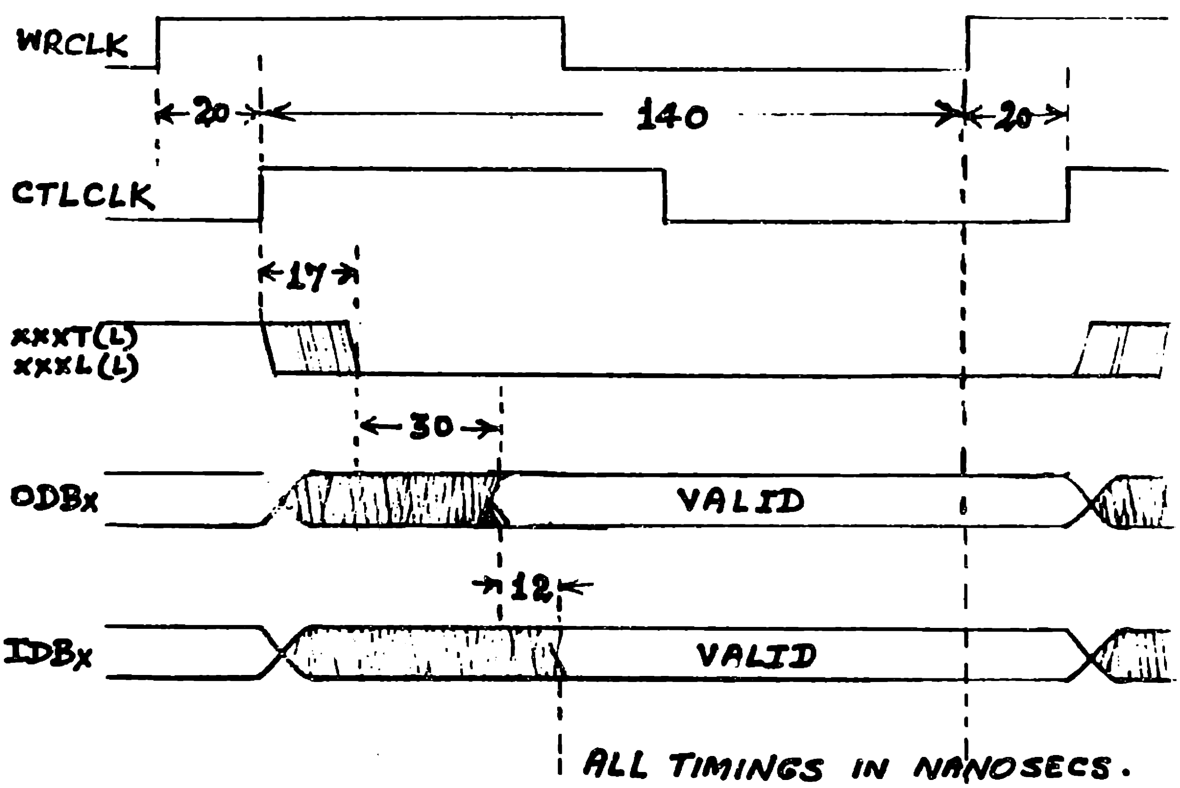


ICM DEBUG STATION

FORWARD TRANSFER



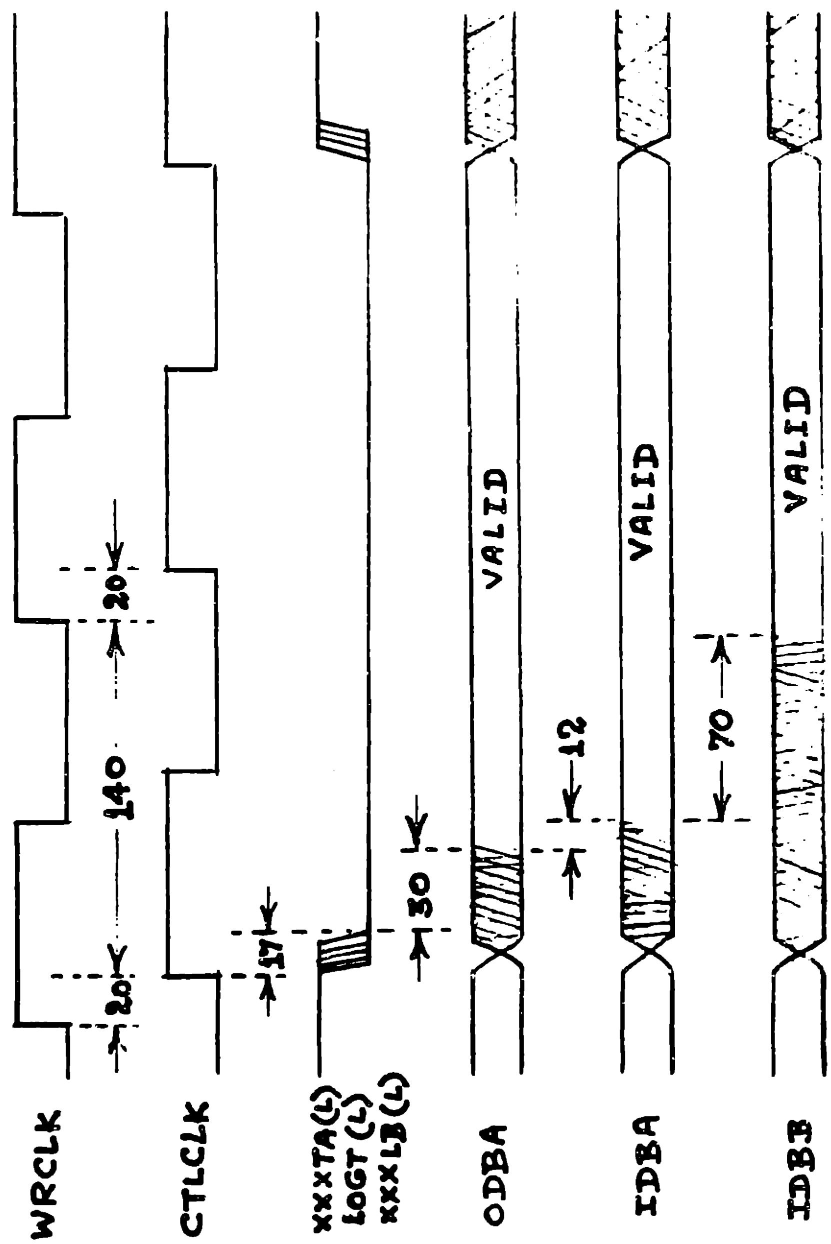
BACKWARD TRANSFER



ALL TIMINGS IN NANOSECS.

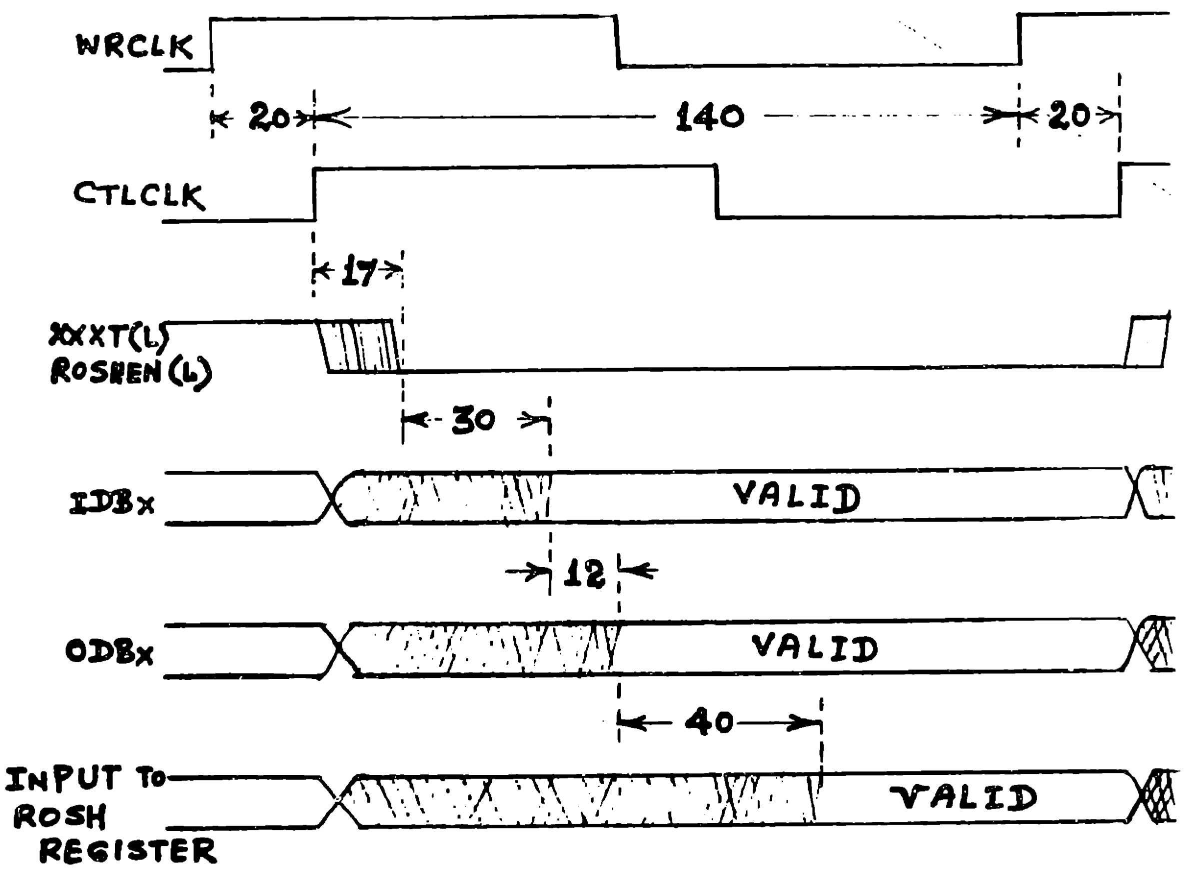
DATA TRANSFER TRANSACTION

NON-LINEAR MAPPING TRANSACTION



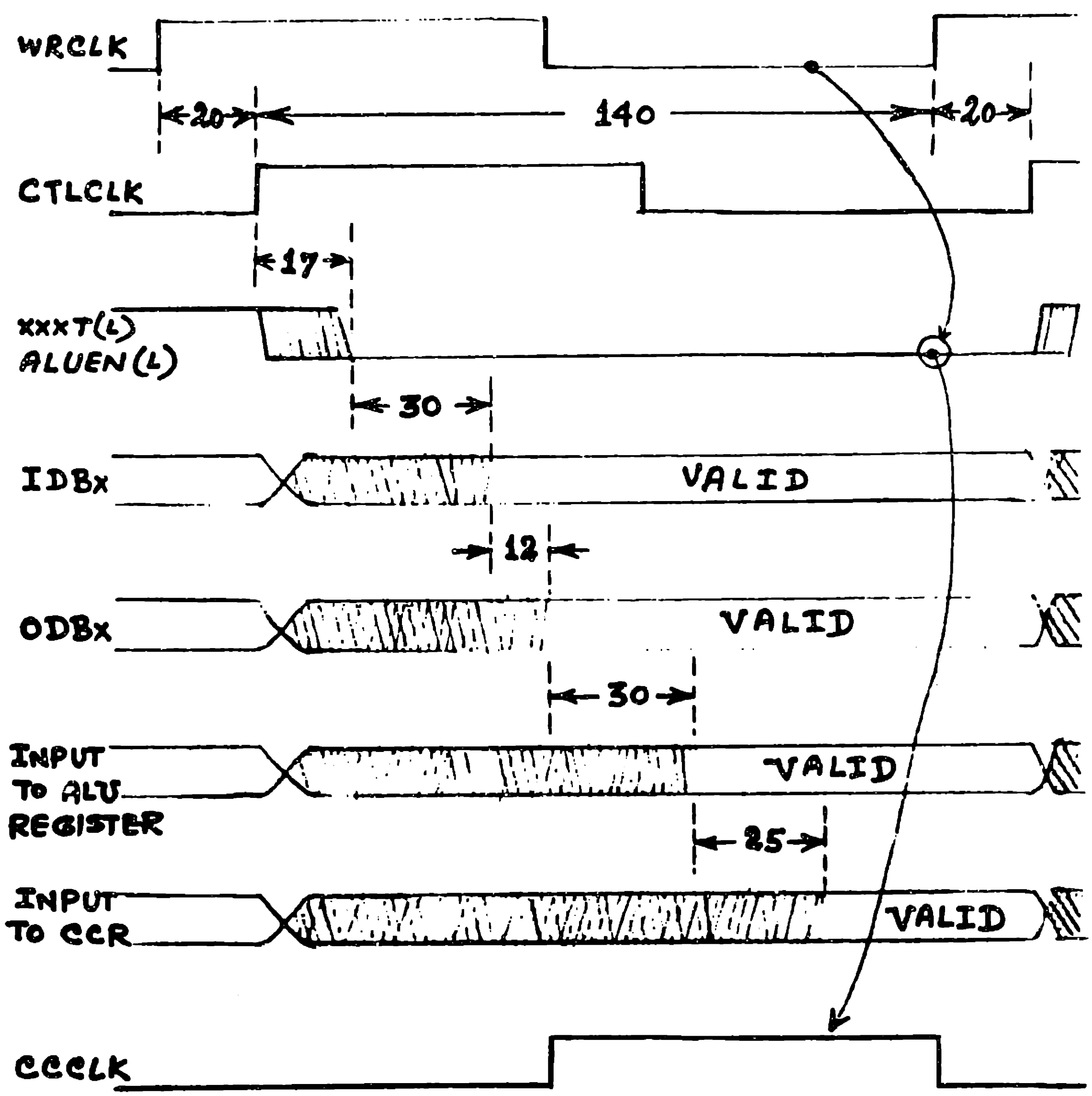
ALL TIMINGS IN NANOSECS.

ROTATE OR SHIFT TRANSACTION



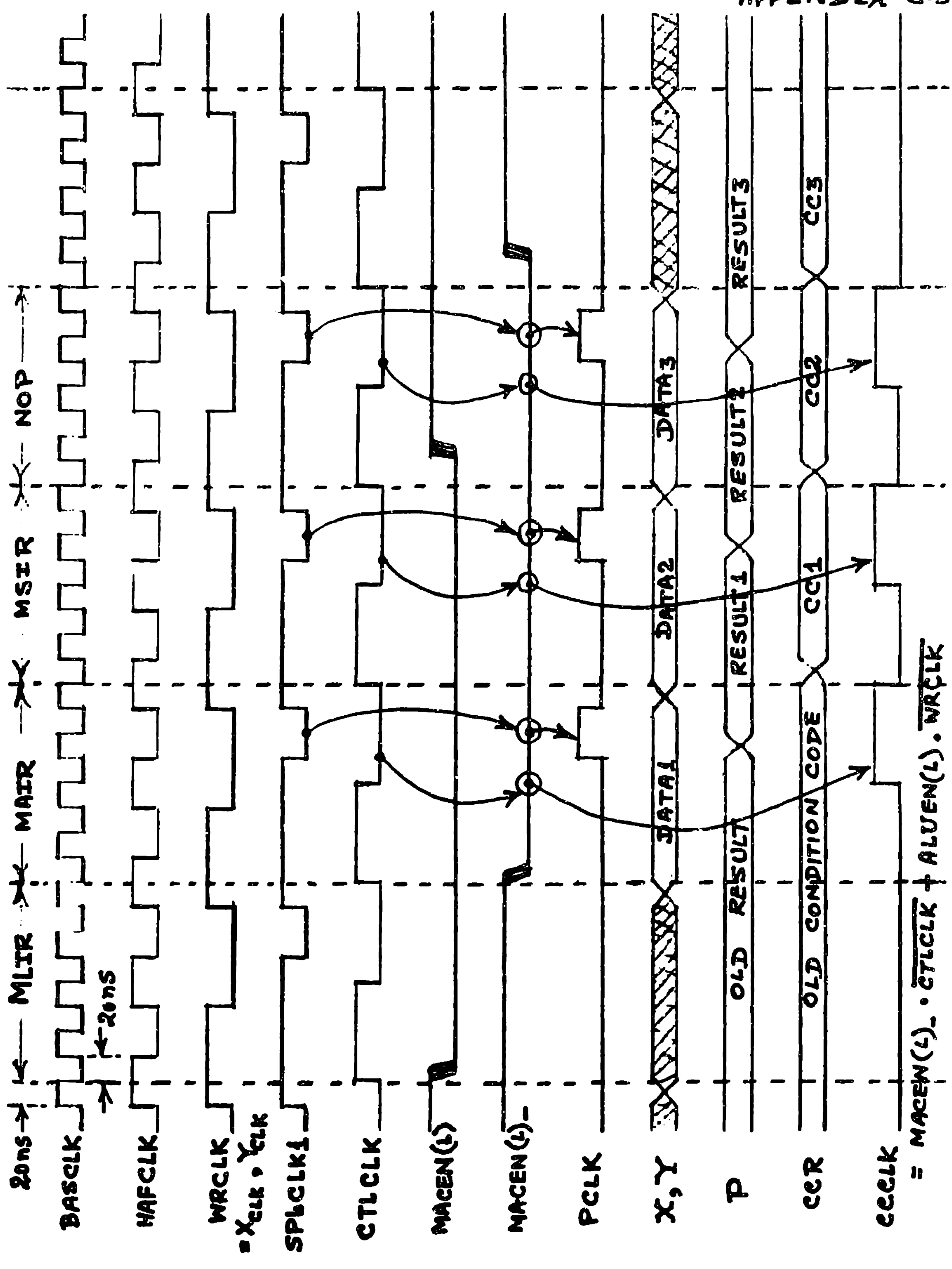
ALL TIMINGS IN NANOSECS.

TIME DIAGRAM - ALU TRANSACTION



$= MACEN(L) \cdot \overline{CTLCLK} + ALUEN(L) \cdot \overline{WRCLK}$

ALL TIMINGS IN NANOSECS.



TIME DIAGRAM - MAC TRANSACTION

RTP's MICROINSTRUCTION FORMAT

63	SPECIAL CONTROL	$\overline{DUSED}$	EU2
62		RTPACK	EV2
61		SPAREC1	ET2
6Ø		SPARECØ	ES2
59	SPARE	SPARE9	EF2
58		SPARE8	EE2
57		SPARE7	ED2
56		SPARE6	DV2
55		SPARE5	DU2
54		SPARE4	DT2
53		SPARE3	DV1
52		SPARE2	DU1
51		SPARE1	DS1
5Ø		SPAREØ	DR1
49	MASK	M1	DB2
48		MØ	DA1
47	IMMEDIATE OPRND	IMD7	ER2
46		IMD6	EP2
45		IMD5	EN2
44		IMD4	EM2
43		IMD3	EL2
42		IMD2	EK2
41		IMD1	EJ2
4Ø		IMDØ	EH2
39	ALE CONTROL	E7	EK1
38		E6	EJ1
37		E5	EH1
36		E4	EF1
35		E3	EE1
34		E2	ED1
33		E1	EC1
32		EØ	EB1

31	SEQUENCE CONTROL	MINST7	EV1
3Ø		MINST6	EU1
29		MINST5	ES1
28		MINST4	ER1
27		MINST3	EP1
26		MINST2	EN1
25		MINST1	EM1
24		MINSTØ	EL1
23	BUS-B CONTROL	BL3	DD2
22		BL2	DE2
21		BL1	DF2
2Ø		BLØ	DH2
19		BED2	DJ2
18		BED1	DK2
17		BEDØ	DL2
16		BT4	DM2
15		BT3	DN2
14		BT2	DP2
13		BT1	DR2
12		BTØ	DS2
11	BUS-A CONTROL	AL3	DB1
1Ø		AL2	DC1
Ø9		AL1	DD1
Ø8		ALØ	DE1
Ø7		AED2	DF1
Ø6		AED1	DH1
Ø5		AEDØ	DJ1
Ø4		AT4	DK1
Ø3		AT3	DL1
Ø2		AT2	DM1
Ø1		AT1	DN1
ØØ		ATØ	DP1

app-D

BUS-A CONTROL

AL3	AL2	AL1	ALØ	AED2	AED1	AEDØ	AT4	AT3	AT2	AT1	ATØ	
X	X	X	X	X	X	X	0	0	0	0	0	- No Talker
X	X	X	X	X	X	X	0	0	0	0	1	- General Reg1 Talk
X	X	X	X	X	X	X	0	0	0	1	0	- General Reg2 Talk
X	X	X	X	X	X	X	0	0	0	1	1	- General Reg3 Talk
X	X	X	X	X	X	X	0	0	1	0	0	- General Reg4 Talk
X	X	X	X	X	X	X	0	0	1	0	1	- General Reg5 Talk
X	X	X	X	X	X	X	0	0	1	1	0	- General Reg6 Talk
X	X	X	X	X	X	X	0	0	1	1	1	- General Reg7 Talk
X	X	X	X	X	X	X	0	1	0	0	0	- General Reg8 Talk
X	X	X	X	X	X	X	0	1	0	0	1	- Immediate Data Reg Talk
X	X	X	X	X	X	X	0	1	0	1	0	- Input Reg1 Talk
X	X	X	X	X	X	X	0	1	0	1	1	- Input Reg2 Talk
X	X	X	X	X	X	X	0	1	1	0	0	- Input Reg3 Talk
X	X	X	X	X	X	X	0	1	1	0	1	
X	X	X	X	X	X	X	0	1	1	1	0	- Reserved For Ring-Buffer
X	X	X	X	X	X	X	0	1	1	1	1	- Reserved For Stack
X	X	X	X	X	X	X	1	0	0	0	0	
X	X	X	X	X	X	X	1	0	0	0	1	- ICT1 Talk
X	X	X	X	X	X	X	1	0	0	1	0	- ICT2 Talk
X	X	X	X	X	X	X	1	0	0	1	2	- ICT3 Talk
X	X	X	X	X	X	X	1	0	1	0	0	- ICT4 Talk
X	X	X	X	X	X	X	1	0	1	0	1	- ICT5 Talk
X	X	X	X	X	X	X	1	0	1	1	0	- ICT6 Talk
X	X	X	X	X	X	X	1	0	1	1	1	- ICT7 Talk
X	X	X	X	X	X	X	1	1	0	0	0	- Reserved For ICT
X	X	X	X	X	X	X	1	1	0	0	1	
X	X	X	X	X	X	X	1	1	0	1	0	- ALU Talk
X	X	X	X	X	X	X	1	1	0	1	1	- MAC-LSP Talk
X	X	X	X	X	X	X	1	1	1	0	0	- MAC-MSP Talk
X	X	X	X	X	X	X	1	1	1	0	1	- Condition Code Reg Talk
X	X	X	X	X	X	X	1	1	1	1	0	- Shifter/Rotator Talk
X	X	X	X	X	X	X	1	1	1	1	1	

app-D

BUS-A CONTROL

AL3	AL2	AL1	ALØ	AED2	AED1	AEDØ	AT4	AT3	AT2	AT1	ATØ	
0	0	0	0	X	X	X	X	X	X	X	X	- NOP
0	0	0	1	X	X	X	X	X	X	X	X	- General Reg1 Listen
0	0	1	0	X	X	X	X	X	X	X	X	- General Reg2 Listen
0	0	1	1	X	X	X	X	X	X	X	X	- General Reg3 Listen
0	1	0	0	X	X	X	X	X	X	X	X	- General Reg4 Listen
0	1	0	1	X	X	X	X	X	X	X	X	- General Reg5 Listen
0	1	1	0	X	X	X	X	X	X	X	X	- General Reg6 Listen
0	1	1	1	X	X	X	X	X	X	X	X	- General Reg7 Listen
1	0	0	0	X	X	X	X	X	X	X	X	- General Reg8 Listen
1	0	0	1	X	X	X	X	X	X	X	X	
1	0	1	0	X	X	X	X	X	X	X	X	
1	0	1	1	X	X	X	X	X	X	X	X	
1	1	0	0	X	X	X	X	X	X	X	X	
1	1	0	1	X	X	X	X	X	X	X	X	
1	1	1	0	X	X	X	X	X	X	X	X	- Reserved For Ring Buffer
1	1	1	1	X	X	X	X	X	X	X	X	- Reserved For Stack

X	X	X	X	0	0	0	X	X	X	X	X	- No Eavesdropper
X	X	X	X	0	0	1	X	X	X	X	X	- Reserved For Ring-Buffer
X	X	X	X	0	1	0	X	X	X	X	X	- Mask Buffer-A Eavesdrop
X	X	X	X	0	1	1	X	X	X	X	X	
X	X	X	X	1	0	0	X	X	X	X	X	
X	X	X	X	1	0	1	X	X	X	X	X	
X	X	X	X	1	1	0	X	X	X	X	X	- Page Buffer Eavesdrop
X	X	X	X	1	1	1	X	X	X	X	X	- MAC-MSP PREL Eavesdrop

BUS-B CONTROL

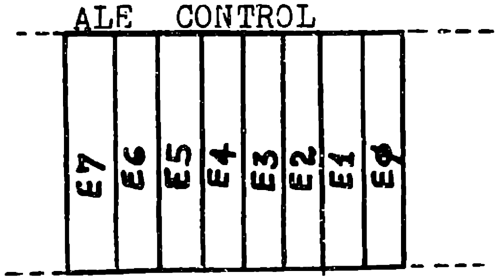
BL3	BL2	BL1	BLØ	BED2	BED1	BEDØ	BT4	BT3	BT2	BT1	BTØ	
X	X	X	X	X	X	X	0	0	0	0	0	- No Talker
X	X	X	X	X	X	X	0	0	0	0	1	- General Reg1 Talk
X	X	X	X	X	X	X	0	0	0	1	0	- General Reg2 Talk
X	X	X	X	X	X	X	0	0	0	1	1	- General Reg3 Talk
X	X	X	X	X	X	X	0	0	1	0	0	- General Reg4 Talk
X	X	X	X	X	X	X	0	0	1	0	1	- General Reg5 Talk
X	X	X	X	X	X	X	0	0	1	1	0	- General Reg6 Talk
X	X	X	X	X	X	X	0	0	1	1	1	- General Reg7 Talk
X	X	X	X	X	X	X	0	1	0	0	0	- General Reg8 Talk
X	X	X	X	X	X	X	0	1	0	0	1	- Immediate Data Reg Talk
X	X	X	X	X	X	X	0	1	0	1	0	- Limit-Low Talk
X	X	X	X	X	X	X	0	1	0	1	1	- Limit-High Talk
X	X	X	X	X	X	X	0	1	1	0	0	- Log Talk
X	X	X	X	X	X	X	0	1	1	0	1	- UCR Talk
X	X	X	X	X	X	X	0	1	1	1	0	- Reserved For Ring-Buffer
X	X	X	X	X	X	X	0	1	1	1	1	- Reserved For Stack
X	X	X	X	X	X	X	1	0	0	0	0	
X	X	X	X	X	X	X	1	0	0	0	1	
X	X	X	X	X	X	X	1	0	0	1	0	
X	X	X	X	X	X	X	1	0	0	1	1	
X	X	X	X	X	X	X	1	0	1	0	0	
X	X	X	X	X	X	X	1	0	1	0	1	
X	X	X	X	X	X	X	1	0	1	1	0	
X	X	X	X	X	X	X	1	0	1	1	1	
X	X	X	X	X	X	X	1	1	0	0	0	
X	X	X	X	X	X	X	1	1	0	0	1	
X	X	X	X	X	X	X	1	1	0	1	0	- ALU Talk
X	X	X	X	X	X	X	1	1	0	1	1	- MAC-LSP Talk
X	X	X	X	X	X	X	1	1	1	0	0	- MAC-MSP Talk
X	X	X	X	X	X	X	1	1	1	0	1	- Condition Code Reg Talk
X	X	X	X	X	X	X	1	1	1	1	0	- Shifter/Rotator Talk
X	X	X	X	X	X	X	1	1	1	1	1	

BUS-B CONTROL

BL3	BL2	BL1	BLØ	BED2	BED1	BEDØ	BT4	BT3	BT2	BT1	BTØ	
0	0	0	0	X	X	X	X	X	X	X	X	- No Listener
0	0	0	1	X	X	X	X	X	X	X	X	- General Reg1 Listen
0	0	1	0	X	X	X	X	X	X	X	X	- General Reg2 Listen
0	0	1	1	X	X	X	X	X	X	X	X	- General Reg3 Listen
0	1	0	0	X	X	X	X	X	X	X	X	- General Reg4 Listen
0	1	0	1	X	X	X	X	X	X	X	X	- General Reg5 Listen
0	1	1	0	X	X	X	X	X	X	X	X	- General Reg6 Listen
0	1	1	1	X	X	X	X	X	X	X	X	- General Reg7 Listen
1	0	0	0	X	X	X	X	X	X	X	X	- General Reg8 Listen
1	0	0	1	X	X	X	X	X	X	X	X	
1	0	1	0	X	X	X	X	X	X	X	X	- Output Reg1 Listen
1	0	1	1	X	X	X	X	X	X	X	X	- Output Reg2 Listen
1	1	0	0	X	X	X	X	X	X	X	X	- Output Reg3 Listen
1	1	0	1	X	X	X	X	X	X	X	X	
1	1	1	0	X	X	X	X	X	X	X	X	- Reserved For Ring-Buffer
1	1	1	1	X	X	X	X	X	X	X	X	- Reserved For Stack
X	X	X	X	0	0	0	X	X	X	X	X	- No Eavesdropper
X	X	X	X	0	0	1	X	X	X	X	X	- Reserved For Ring-Buffer
X	X	X	X	0	1	0	X	X	X	X	X	- Mask Buffer-B Eavesdrop
X	X	X	X	0	1	1	X	X	X	X	X	- ICT ADDR Reg1 Eavesdrop
X	X	X	X	1	0	0	X	X	X	X	X	- ICT ADDR Reg2 Eavesdrop
X	X	X	X	1	0	1	X	X	X	X	X	- ICT ADDR Reg3 Eavesdrop
X	X	X	X	1	1	0	X	X	X	X	X	- MAC-XTP PREL Eavesdrop
X	X	X	X	1	1	1	X	X	X	X	X	- MAC-LSP PREL Eavesdrop

app-D

ALE CONTROL



E7	E6	E5	E4	E3	E2	E1	E0	Function	Group
0	0	X	X	X	X	X	X	NOP	
0	1	0	X	X	X	X	1	Multiply-Accumulate	MAC
0	1	0	X	X	X	1	X	Multiply-Subtract	MAC
0	1	0	X	X	1	X	X	Round-Off/Truncate	MAC
0	1	0	X	1	X	X	X	Two's-Complement	MAC
0	1	1	X	X	0	0	0	XTP=Hiz, MSP=Hiz, LSP=Hiz	MAC
0	1	1	X	X	0	0	1	XTP=Hiz, MSP=Hiz, LSP=PL	MAC
0	1	1	X	X	0	1	0	XTP=Hiz, MSP=PL, LSP=Hiz	MAC
0	1	1	X	X	0	1	1	XTP=Hiz, MSP=PL, LSP=PL	MAC
0	1	1	X	X	1	0	0	XTP=PL, MSP=Hiz, LSP=Hiz	MAC
0	1	1	X	X	1	0	1	XTP=PL, MSP=Hiz, LSP=PL	MAC
0	1	1	X	X	1	1	0	XTP=PL, MSP=PL, LSP=Hiz	MAC
0	1	1	X	X	1	1	1	XTP=PL, MSP=PL, LSP=PL	MAC
1	0	X	X	X	X	X	1	Function Select S0	ALU
1	0	X	X	X	X	1	X	Function Select S1	ALU
1	0	X	X	X	1	X	X	Function Select S2	ALU
1	0	X	X	1	X	X	X	Function Select S3	ALU
1	0	X	1	X	X	X	X	Carry Input	ALU
1	0	1	X	X	X	X	X	Function Mode	
1	1	X	X	X	0	0	0	Byte Shift Left	ROSH
1	1	X	X	X	0	0	1	Byte Shift Right	ROSH
1	1	X	X	X	0	1	0	Byte Rotate Left	ROSH
1	1	X	X	X	0	1	1	Byte Rotate Right	ROSH
1	1	X	X	X	1	0	0	Word Shift Left	ROSH
1	1	X	X	X	1	0	1	Word Shift Right	ROSH
1	1	X	X	X	1	1	0	Word Rotate Left	ROSH
1	1	X	X	X	1	1	1	Word Rotate Right	ROSH

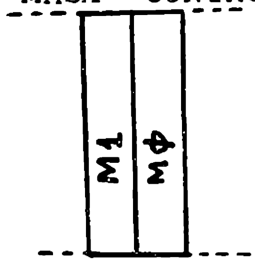
app-D

SEQUENCE CONTROL

MINST7	MINST6	MINST5	MINST4	MINST3	MINST2	MINST1	MINST⌀	
0	0	0	0	X	X	X	X	- ALU Zero/MAC Bit 15
0	0	1	0	X	X	X	X	- ALU Minus/MAC Bit 16
0	1	0	0	X	X	X	X	- ALU Logical Carry/MAC Bit 18
0	1	1	0	X	X	X	X	- ALU Hardware Carry/MAC Bit 17
1	0	0	0	X	X	X	X	- ALU Equal/MAC XTP Minus
1	0	1	0	X	X	X	X	- ALU Overflow/MAC Bit 14
1	1	0	0	X	X	X	X	- RTP New-Data
1	1	1	0	X	X	X	X	- Manager's Flag
X	X	X	1	X	X	X	X	- CCEN_Unconditional
X	X	X	X	0	0	0	0	- JZ Jump Zero
X	X	X	X	0	0	0	1	- CJS Cond JSB PL
X	X	X	X	0	0	1	0	- JMAP Jump Map
X	X	X	X	0	0	1	1	- CJP Cond Jump PL
X	X	X	X	0	1	0	0	- PUSH Push/Cond Load CTR
X	X	X	X	0	1	0	1	- JSRP Cond JSB R/PL
X	X	X	X	0	1	1	0	- CJV Cond Jump Vector
X	X	X	X	0	1	1	1	- JRP Cond Jump R/PL
X	X	X	X	1	0	0	0	- RFCT Repeat Loop, CNTR=0
X	X	X	X	1	0	0	1	- RPCT Repeat PL, CNTR=0
X	X	X	X	1	0	1	0	- CRTN Cond Return
X	X	X	X	1	0	1	1	- CJPP Cond Jump PL & POP
X	X	X	X	1	1	0	0	- LDCT LD CNTR & Continue
X	X	X	X	1	1	0	1	- LOOP Test End Loop
X	X	X	X	1	1	1	0	- CONT Continue
X	X	X	X	1	1	1	1	- TWB Three-Way Branch

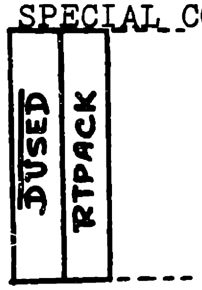
app-D

MASK CONTROL



0 0 - Disable Both Masks
0 1 - Enable Mask-A Only
1 0 - Enable Mask-B Only
1 1 - Enable Both Masks

SPECIAL CONTROL



0 X - Reset Data Flag
X 1 - Set Acknowledge Flag

APPENDIX - E

```
;        INK  CORRECTION  MONITOR
;
;             VERSION 1.1
;
        .TITLE  ICMON-V1.1
;
;  This software system is a COMMAND INTERPRETER, intended for
;  operation on a PDP-11 computer, to facilitate management of
;  the INK CORRECTION MODULE. It accepts commands, which may
;  be upto 4 characters long, from the console terminal and
;  interacts with ICM to perform various initialization or
;  diagnostic functions within the ICM. The interaction
;  occurs by virtue of handshaking between this Interpreter
;  and the ICM SUPERVISOR (another software system, resident
;  in a Microcomputer named ICM CONTROLLER) through special
;  hardware called UNIBUS INTERFACE. This Interface maps ICM
;  Manager in the Unibus address space as two I/O ports
;  viz. Control Port (ICMCSR, Address=164152) and Data
;  Port (ICMBUF, Address=164150). From the Unibus end, a
;  command may be written into the Control Port or status
;  may be read from it. If the Control Port is written into
;  the ICM Manager responds, through its interrupt system,
;  by activating appropriate service routine. Data exchange
;  occurs through the Data Port.
;
;        EDIT  HISTORY
;
;JUL-23-80      SNM     Original.
;OCT-16-80      SNM     LDPT Command added.
;
;
;        REGISTER EQUATES
;
R0 = %0
R1 = %1
R2 = %2
R3 = %3
R4 = %4
R5 = %5
SP = %6
PC = %7
;
;
;        I/O EQUATES
;
```

```
;   Console Terminal
CTRCSR = 177560          ;receiver control & status reg
CTRBUF = 177562          ;receiver data buffer
CTXCSR = 177564          ;transmitter control & status reg
CTXBUF = 177566          ;transmitter data buffer
;
;   ICM Interface
ICMCSR = 164152          ;ICM control port
ICMBUF = 164150          ;ICM data port
;
;
;       PROGRAM EQUATES
;
;       None.
;
;
;       MACRO DEFINITIONS
;
;
        .MACRO  MULRX3,RX         ;multiplies named reg by 3
        MOV     RX,-(SP)          ;save regiseter on stack
        ASL     RX                ;multiply first by 2
        ADD     (SP)+,RX          ;then add once from stack
        .ENDM
;
;
;
        .MACRO  GETARG,PARM,LENTH,DEST  ;fetches named param
        JSR     R5,WRITE          ;from console terminal & loads
        .WORD   PARM              ;it in named destination
        .WORD   LENTH             ;LENTH = parameter name length
        .WORD   IOBUFC
        JSR     R5,PUTEQU
        JSR     R5,GETNUM         ;uses NUMBFR thru GETNUM subr.
        MOV     NUMBFR,DEST
        .ENDM
;
;
;
        .ASECT                    ;program begins here
;
        .=1000
        MOV     PC,SP             ;init stack pointer
        TST     -(SP)             ;dummy push to init stack ptr.
        MOV     #TRHAND,@#4       ;load trap vector
        MOV     #074340,@#6
        MOV     #6,R0             ;space for program header
MONHDR: JSR     R5,PUTSKP         ;skip to next line
        DEC     R0                ;check if done
        BNE     MONHDR            ;no, do it again
        JSR     R5,PUTAB          ;put a Tab
        JSR     R5,WRITE          ;write top line of header
        .WORD   MSHDR1
        .WORD   BIN080
```

app-E

```
        .WORD   IOBUFC
        JSR     R5,PUTSKP
        JSR     R5,PUTAB
        JSR     R5,WRITE              ;write 2nd line of header
        .WORD   MSHDR2
        .WORD   BIN020
        .WORD   IOBUFC
        JSR     R5,PUTVER             ;write ICMS version number
        JSR     R5,PUTSKP             ;skip to next line
PROCMD: JSR     R5,PUTSKP             ;start a new line
        JSR     R5,WRITE              ;write prompt "COMMAND := "
        .WORD   MSGCMD
        .WORD   BIN007
        .WORD   IOBUFC
        JSR     R5,PUTEQU
        JSR     R5,GETCMD             ;get a command from terminal
        CLR     R0                    ;R0 will contain command code
        JSR     R5,GOCMD              ;go, process fetched command
        BR      PROCMD                ;loop until EXIT command
;
;
;
;       GET-CoMmanD Routine
;
;   This routine fetches a command word, which can be upto 4
;   characters long, and returns it in IOBUF. If operator types
;   more than 4 characters then only last 4 characters are
;   returned. If fewer than 4 characters are typed, then the
;   string is padded with tailing spaces. IOBUFC keeps
;   a count of of actual number of characters typed in.
;
;       Calling Sequence:
;                                     JSR     R5,GETCMD
;
;
GETCMD: JSR     R5,READ               ;read from console terminal
        .WORD   IOBUF
        .WORD   BIN004
        .WORD   IOBUFC
        MOV     IOBUFC,R0             ;check # of characters read
GETC1:  CMP     R0,#4                 ;4 characters read?
        BGE     GETC2                 ;yes, bypass padding
        MOVB    SPACE,IOBUF(R0)       ;no, pad with a 'space'
        INC     R0                    ;increment # of chars now
        BR      GETC1                 ;go back and check if done
GETC2:  RTS     R5
;
;
;
;       GO-CoMmanD Routine
;
;   This routine matches the string fetched in IOBUF with
;   the entries in the Command List. If a match is found,
;   control is transferred to the corresponding process
```

```
;  routine. If no match is found, exit is made via
;  NOCMD routine.
;
;        Calling Sequence:
;                                        JSR      R5,GOCMD
;
;
GOCMD:   MOV      R0,R1                 ;init R1 pointing to beginning
         MULRX3   R1                    ;3X,list has 3 word entries
         ASL      R1                    ;2X, to get 6 bytes offset
         TSTB     CMDLST(R1)            ;check if end of list
         BEQ      NOCMD                 ;yes, then exit
         CMP      IOBUF,CMDLST(R1) ;no, compare 1st 2 characters
         BNE      GOCM1                 ;no match, go for next in list
         CMP      IOBUF+2,CMDLST+2(R1) ;matched, compare next 2
         BNE      GOCM1                 ;no match, go for next in list
         JMP      @CMDLST+4(R1)         ;matched, goto corres. routine
GOCM1:   INC      R0                    ;point to next entry in list
         BR       GOCMD                 ;repeat search till done
;
;
;
;        NOt-valid CoMmanD Routine
;
;  Prints message warning that an invalid command
;  name was typed.
;
;
NOCMD:   JSR      R5,PUTQOT             ;write a double quote
         JSR      R5,WRITE              ;write command word as read
         .WORD    IOBUF
         .WORD    BIN004
         .WORD    IOBUFC
         JSR      R5,PUTQOT             ;write another double quote
         JSR      R5,WRITE              ;write message 1
         .WORD    MSGNC1
         .WORD    BIN080
         .WORD    IOBUFC
         JSR      R5,WRITE              ;write message 2
         .WORD    MSGNC2
         .WORD    BIN080
         .WORD    IOBUFC
         RTS      R5
;
;
;
;        HELP COMMAND PROCESS ROUTINE
;
;  Prints a list of valid commands.
;
;
HELP:    JSR      R5,WRITE              ;write top message
         .WORD    MSGHLP
         .WORD    BIN080
```

```
        .WORD   IOBUFC
        ADD     #CMDLST,R0          ;offset address of comm. word
HLP1:   TSTB    (R0)                ;check if done
        BEQ     HLP2                ;yes, then exit
        JSR     R5,PUTAB            ;no, put space first
        JSR     R5,PUTAB
        JSR     R5,PUTAB
        MOV     (R0)+,IOBUF         ;transfer 1st 2 chars
        MOV     (R0)+,IOBUF+2       ;transfer next 2 chars
        JSR     R5,WRITE            ;write the pointed comm. word
        .WORD   IOBUF
        .WORD   BIN004
        .WORD   IOBUFC
        JSR     R5,PUTSKP           ;skip to next line
        TST     (R0)+               ;skip next word in command list
        BR      HLP1                ;repeat process till done
HLP2:   RTS     R5
;
;
;
;       GET COMMAND PROCESS ROUTINE
;
;  Transfers a record of specified length from specified
;  source address in ICM to the specified destination
;  in HOst Processor.
;
;
;
GET:    GETARG  MSADDR,BIN014,R1    ;get source address in R1
        GETARG  MDADDR,BIN019,R2    ;get dest. address in R2
        GETARG  MWDCNT,BIN010,R3    ;get word count in R3
GET1:   TST     R3                  ;check if word count = 0
        BLE     GET5                ;yes, exit
        CMP     #376,R3             ;no, check if word count < 376
        BLE     GET2                ;no, transfer 376 words first
        MOV     R3,R4               ;yes, transfer just as many
        BR      GET3                ;proceed for transfer
GET2:   MOV     #376,R4             ;376 words to be transferred
GET3:   MOV     R1,ICMDTA           ;load source address
        JSR     R5,PUTICM           ;transfer data to ICM
        .WORD   ICMDTA
        SWAB    R4                  ;align for qualifier field
        BIS     R4,R0               ;set qualifier
        SWAB    R4                  ;reset length parameter
GDELAY: BIT     #40,@#ICMCSR        ;ICMC busy?
        BNE     GDELAY              ;yes, then loop
        MOV     R0,@#ICMCSR         ;no, write new control word
        BIC     #177400,R0          ;blank qualifier field
GET4:   JSR     R5,GETICM           ;transfer data from ICM
        .WORD   ICMDTA
        MOV     ICMDTA,(R2)+        ;store in dest., point next
        ADD     #2,R1               ;update source address
        DEC     R3                  ;update final transfer count
        DEC     R4                  ;update current transfer count
        BNE     GET4                ;loop, if transfer not finished
```

```
        BR      GET1                 ;loop till full transfer done
GET5:   RTS     R5
;
;
;
;       PUT COMMAND PROCESS ROUTINE
;
;  Transfers a record of specified length from specified source
;  address in HOP to the specified destination in ICM.
;
;
PUT:    GETARG  MSADDR,BIN014,R1   ;get source address in R1
        GETARG  MDADDR,BIN019,R2   ;get dest. address in R2
        GETARG  MWDCNT,BIN010,R3   ;get word count in R3
PUT1:   TST     R3                 ;check if word count = 0
        BLE     PUT5               ;yes, exit
        CMP     #376,R3            ;no, check if word count <376
        BLE     PUT2               ;no, transfer 376 words first
        MOV     R3,R4              ;yes, transfer just as many
        BR      PUT3               ;proceed for transfer
PUT2:   MOV     #376,R4            ;376 words to be transferred
PUT3:   MOV     R2,ICMDTA          ;load destination address
        JSR     R5,PUTICM          ;transfer data to ICM
        .WORD   ICMDTA
        SWAB    R4                 ;align for qualifier field
        BIS     R4,R0              ;set qualifier
        SWAB    R4                 ;reset length parameter
PDELAY: BIT     #40,@#ICMCSR       ;ICMC busy?
        BNE     PDELAY             ;yes, loop
        MOV     R0,@#ICMCSR        ;no, write a new control word
        BIC     #177400,R0         ;blank qualifier field
PUT4:   MOV     (R1)+,ICMDTA       ;load from source, point next
        ADD     #2,R2              ;update destination address
        JSR     R5,PUTICM          ;transfer data to ICM
        .WORD   ICMDTA
        DEC     R3                 ;update final transfer count
        DEC     R4                 ;update current transfer count
        BNE     PUT4               ;loop, if transfer not finished
        BR      PUT1               ;loop till full transfer done
PUT5:   RTS     R5
;
;
;
;       Error-DuMP COMMAND PROCESS ROUTINE
;
;  Fetches ERROR Code from ICM and prints error message,
;
;
EDMP:   BIT     #40,@#ICMCSR       ;ICMC busy?
        BNE     EDMP               ;yes, loop
        MOV     R0,@#ICMCSR        ;no, write a new control word
        JSR     R5,GETICM          ;transfer data from ICM
        .WORD   ICMDTA
        MOV     #5,R0              ;error message = 32 chars max.
```

```
EDMP1:  ASL     ICMDTA              ;multiply error code by 32
        DEC     R0
        BNE     EDMP1
        ADD     #ERRTBL,ICMDTA      ;add offset to Table Origin
        MOV     ICMDTA,ERRMSG       ;pass parameter to WRITE subr.
        JSR     R5,WRITE            ;go write message
ERRMSG: .WORD   ERRTBL
        .WORD   BIN032
        .WORD   IOBUFC
        RTS     R5
;
;
;
;    Error-ReSeT, (RTP)RUN, (RTP)HOLD COMMAND PROCESS ROUTINES
;
;  ERST process resets RTP error state.
;  RUN process sets RTP in run mode.
;  HOLD process freezes RTP in hold mode.
;  DIGx processes perform diagnostics.
;  All these RTP processes are activated by simply writing
;  appropriate control word into ICM Control port.
;
ERST:
RUN:
DIG0:
DIG1:
DIG2:
DIG3:
DIG4:
DIG5:
DIG6:
DIG7:
HOLD:   BIT     #40,@#ICMCSR        ;ICMC busy?
        BNE     HOLD                ;yes, loop
        MOV     R0,@#ICMCSR         ;no, write a new control word
        RTS     R5
;
;
;
;       Status-DuMP COMMAND PROCESS ROUTINE
;
;  Fetches ICM status, decodes it and prints appropriate
;  status messages.
;
;
SDMP:   BIT     #40,@#ICMCSR        ;ICMC busy?
        BNE     SDMP                ;yes, loop
        MOV     R0,@#ICMCSR         ;no, write a new control word
        JSR     R5,GETICM           ;transfer data from ICM
        .WORD   ICMDTA
        MOV     ICMDTA,R1           ;start decoding color mode
        BIC     #177774,R1          ;balnk other bits
        MOV     #5,R2               ;status messages = 32 chars max.
SDMP1:  ASL     R1                  ;multiply code by 32
```

```
        DEC     R2
        BNE     SDMP1
        ADD     #STBL1,R1       ;add offset to STBL1 origin
        MOV     R1,STMSG1       ;pass as parameter to WRITE subr.
        JSR     R5,WRITE        ;go write color mode
STMSG1: .WORD   STBL1
        .WORD   BIN032
        .WORD   IOBUFC
        MOV     #4,R1           ;start decoding turn-off bits
        CLR     R3              ;clear flag for inhibit bits
SDMP2:  CMP     #4,R2           ;all 4 turn-off bits checked?
        BLE     SDMP5           ;yes, go next
        BIT     R1,ICMDTA       ;no, check current bit
        BEQ     SDMP4           ;bit not set, go for next bit
        MOV     R2,R3           ;bit set, then first set flag
        MOV     #5,R4           ;message = 32 characters max.
SDMP3:  ASL     R3              ;multiply code by 32
        DEC     R4
        BNE     SDMP3
        ADD     #STBL2,R3       ;add offset to STBL2 origin
        MOV     R3,STMSG2       ;pass as parameter to WRITE subr.
        JSR     R5,WRITE        ;go write message
STMSG2: .WORD   STBL2
        .WORD   BIN032
        .WORD   IOBUFC
SDMP4:  ASL     R1              ;shift template for next bit
        INC     R2              ;update count
        BR      SDMP2           ;loop until all bits checked
SDMP5:  TST     R3              ;test flag
        BNE     SDMP6           ;flag set, skip to next step
        MOV     #200,R4         ;not set, another message
        ADD     #STBL2,R4       ;add offset to STBL2 origin
        MOV     R4,STMSG3       ;pass as parameter to WRITE subr.
        JSR     R5,WRITE        ;go write message
STMSG3: .WORD   STBL2
        .WORD   BIN032
        .WORD   IOBUFC
SDMP6:  MOV     ICMDTA,R0       ;start decoding Operation mode
        BIC     #177477,R0      ;blank other bits
        ASR     R0              ;compute offset corres. to code
        ADD     #STBL3,R0       ;add offset to STBL3 origin
        MOV     R0,STMSG4       ;pass as parameter to WRITE subr.
        JSR     R5,WRITE        ;go write operation mode
STMSG4: .WORD   STBL3
        .WORD   BIN032
        .WORD   IOBUFC
        RTS     R5
;
;
;
;
;       CoMPaRe COMMAND PROCESS ROUTINE
;
;
; Compares two records of specified length at specified
; addresses in HOP. In case of exact match, signals 0 error.
```

```
;  Else, prints addresses and their contents wherever a match
;  did not occur and finally gives number of errors.
;
;
CMPR:   GETARG  MSADDR,BIN014,R1   ;get source address in R1
        GETARG  MDADDR,BIN019,R2   ;get dest. address in R2
        GETARG  MWDCNT,BIN010,R3   ;get word count in R3
        CLR     R0                 ;R0 will keep count of errors
CMPR1:  TST     R3                 ;test if done
        BLE     CMPR2              ;yes, go exit procedure
        DEC     R3                 ;no, update word count
        CMP     (R1)+,(R2)+        ;compare words & point next
        BEQ     CMPR1              ;matched, loop till done
        CMP     -(R1),-(R2)        ;no match, point back to it
        JSR     R5,WRITE           ;write "source address"
        .WORD   MSADDR
        .WORD   BIN014
        .WORD   IOBUFC
        JSR     R5,PUTEQU          ;write ":="
        MOV     R1,NUMBFR          ;load source address
        JSR     R5,NUMDMP          ;write it
        JSR     R5,PUTAB           ;write a tab char
        JSR     R5,WRITE           ;write "contents"
        .WORD   MSCONT
        .WORD   BIN008
        .WORD   IOBUFC
        JSR     R5,PUTEQU          ;write ":="
        MOV     (R1)+,NUMBFR       ;load pointed word & point next
        JSR     R5,NUMDMP          ;write it
        JSR     R5,PUTSKP          ;skip to next line
        JSR     R5,WRITE           ;write "destination address"
        .WORD   MDADDR
        .WORD   BIN019
        .WORD   IOBUFC
        JSR     R5,PUTEQU          ;write ":="
        MOV     R2,NUMBFR          ;load destination address
        JSR     R5,NUMDMP          ;write it
        JSR     R5,PUTAB           ;write a tab char
        JSR     R5,WRITE           ;write "contents"
        .WORD   MSCONT
        .WORD   BIN008
        .WORD   IOBUFC
        JSR     R5,PUTEQU          ;write ":="
        MOV     (R2)+,NUMBFR       ;load pointed word & point next
        JSR     R5,NUMDMP          ;write it
        JSR     R5,PUTSKP          ;skip to next line
        INC     R0                 ;increment number of errors
        BR      CMPR1              ;loop till done
CMPR2:  MOV     R0,NUMBFR          ;load number of errors
        JSR     R5,PUTNUM          ;write it
        JSR     R5,WRITE           ;write "errors detected"
        .WORD   MERDET
        .WORD   BIN020
        .WORD   IOBUFC
```

app. E

```
        RTS     R5
;
;
;
;       INIT COMMAND PROCESS ROUTINE
;
;  Performs initialization of the Real-time Processor.
;  Two major actions taken are downloading of Microprogram and
;  Ink Correction Table corresponding to the Color Mode, which
;  is defined by the current status.
;
;
INIT:   MOV     R0,-(SP)        ;save command code for later
IDELAY: BIT     #40,@#ICMCSR    ;ICMC busy?
        BNE     IDELAY          ;yes, loop
        MOV     #7,@#ICMCSR     ;no, write control word
        JSR     R5,GETICM       ;transfer data from ICM
        .WORD   ICMDTA
        BIC     #177774,ICMDTA  ;isolate status code
        MOV     #12,R4          ;compute offset into the tables
INIT1:  ASL     ICMDTA          ;X by record length (2Kbytes)
        DEC     R4
        BNE     INIT1
        MOV     ICMDTA,-(SP)    ;save offset for next download
        TST     ICMDTA          ;color mode = Black?
        BEQ     INIT2           ;yes, no ICT be downloaded
        MOV     #2,R0           ;no, set control word
        MOV     #1000,R3        ;set record length = 1Kwords
        MOV     #INKTBL,R1      ;load INKTBL origin
        ADD     ICMDTA,R1       ;add offset
        SUB     #2000,R1        ;shift origin to align record
        MOV     #104000,R2      ;set destination address in ICM
        JSR     R5,PUT1         ;go download
INIT2:  MOV     #2,R0           ;set control word for download
        MOV     #770,R3         ;set record length (770 octal)
        MOV     #MICPGM,R1      ;load MICPGM origin
        ADD     (SP)+,R1        ;add offset from stack
        MOV     #100020,R2      ;set dest. address in ICM
        JSR     R5,PUT1         ;go download
        MOV     (SP)+,@#ICMCSR  ;write cont. word for RTP init
        RTS     R5
;
;
;
;
;       Octal-DuMP COMMAND PROCESS ROUTINE
;
;  Performs a core-dump, in octal, of a specified length
;  record from a specified destination in HOP.
;
;
ODMP:   GETARG  MSADDR,BIN014,R1  ;get source address in R1
        GETARG  MWDCNT,BIN010,R2  ;get record length in R2
ODMP1:  JSR     R5,PUTSKP         ;skip to next line
```

```
        TST     R2                      ;word count = 0?
        BLE     ODMP4                   ;yes, exit
        CMP     #10,R2                  ;no, word count > 8
        BLE     ODMP2                   ;yes, first process 8 words
        MOV     R2,R3                   ;no, process just as many
        BR      ODMP3
ODMP2:  MOV     #10,R3                  ;set loop count = 8
ODMP3:  JSR     R5,PUTSPC               ;write a space char
        JSR     R5,PUTSPC               ;write another space
        MOV     (R1)+,NUMBFR            ;load mem. contents, point next
        JSR     R5,NUMDMP               ;dump in octal
        DEC     R2                      ;update word count
        DEC     R3                      ;update loop count
        BNE     ODMP3                   ;loop till 8 words are dumped
        BR      ODMP1                   ;loop till full record dumped
ODMP4:  RTS     R5
;
;
;
;       LOAD COMMAND PROCESS ROUTINE
;
;  Loads specified number of data words from console
;  terminal at the specified address in HOP.
;
;
LOAD:   GETARG  MDADDR,BIN019,R1  ;get dest. address in R1
        GETARG  MWDCNT,BIN010,R2  ;get word count in R2
LOD1:   TST     R2                      ;word count = 0?
        BLE     LOD2                    ;yes, then exit
        JSR     R5,GETNUM               ;no, get a word from terminal
        MOV     NUMBFR,(R1)+            ;store & point next
        DEC     R2                      ;update word count
        BR      LOD1                    ;loop till done
LOD2:   RTS     R5
;
;
;
;       LOAD PAPER-TAPE COMMAND PROCESS ROUTINE
;
;  Loads specified length reecord from paper-tape reader of
;  the Console terminal at the specified address.
;
LDPT:   GETARG  MDADDR,BIN019,R1  ;get destination address
        GETARG  MWDCNT,BIN010,R2  ;get word count
        ASL     R2                      ;X by 2 to get byte count
LDP1:   TST     R2                      ;done?
        BLE     LDP3                    ;yes, exit
        INCB    @#CTRCSR                ;not done, go transfer a byte
LDP2:   TSTB    @#CTRCSR                ;receiver busy?
        BPL     LDP2                    ;yes, loop
        MOVB    @#CTRBUF,(R1)+          ;no, transfer a byte
        DEC     R2                      ;update byte count
        BR      LDP1                    ;repeat till done
LDP3:   RTS     R5
```

app-E

```
;
;
;
;       EXIT COMMAND PROCESS ROUTINE
;
;       Terminates Command Interpreter ICMON.
;
;
EXIT:   HALT                            ;hang up
;
;
;
;       SUB-ROUTINE LIBRARY
;
;
;   READ Sub-routine
;
;   This sub-routine reads from the console terminal.
;   The address of the input buffer, the address of the
;   buffer length argument, and the address of the byte
;   count buffer are passed as parameters. All characters
;   read are immediately echoed to the terminal. The
;   process is terminated when a Carriage-return character
;   is read, whereby the control is transferred to the
;   caller of the sub-routine. Only a string of specified
;   length is returned in the specified input buffer.
;   If more characters are typed, the buffer is scrolled
;   so that only last typed as many characters are returned.
;   The number of bytes actually read from the terminal
;   is returned in CNTBUF.
;
;   Calling Sequence:
;       JSR     R5,READ
;       .WORD   INBUF           address of input buffer
;       .WORD   LENBUF          address of length argument
;       .WORD   CNTBUF          address of byte count buffer
;
;   Return Conditions:
;       Number of bytes read from terminal is stored in COUNT.
;   Symbolic References:
;       CTXCSR, CTXBUF, CTRCSR, CTRBUF, CRET, LFEED
;   Register Usage:
;                       R0
;                       R1
;                       R2
;                       R3
;
READ:   MOV     R0,-(SP)                ;save caller's registers
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     (R5)+,R0                ;get buffer address in R0
        MOV     @(R5),R1                ;get length count in R1
        ADD     R0,R1                   ;offset address by length
```

```
RD1:    CLR     -(R1)           ;zap buffer for spec. length
        CMP     R1,R0
        BNE     RD1
        CLR     R1              ;zap length count
RD2:    TSTB    @#CTRCSR        ;receiver busy?
        BPL     RD2             ;yes, loop
        MOVB    @#CTRBUF,R3     ;no, read character
RD3:    TSTB    @#CTXCSR        ;transmitter busy?
        BPL     RD3             ;yes, loop
        MOVB    R3,@#CTXBUF     ;no, echo character
        BICB    #200,R3         ;clear parity bit
        CMPB    CRET,R3         ;carriage return character?
        BEQ     RD6             ;yes, end of line, exit
        CMP     R1,@(R5)        ;no, all bytes read?
        BGE     RD4             ;yes, shift buffer as FIFO
        MOVB    R3,(R0)+        ;no, store character in buffer
        INC     R1              ;increment length count
        BR      RD2             ;go for next character
RD4:    MOV     R3,-(SP)        ;save last character on stack
        MOV     R0,R3           ;copy end address of buffer
        SUB     @(R5),R3        ;subtr. length for start addr
RD5:    MOVB    1(R3),(R3)+     ;shift buffer as FIFO
        CMP     R0,R3           ;check if done
        BNE     RD5             ;no, loop till done
        MOV     (SP)+,R3        ;restore char from stack now
        MOVB    R3,-1(R0)       ;store at the top of buffer
        INC     R1              ;increment # of chars typed
        BR      RD2             ;go for next character
RD6:    TSTB    @#CTXCSR        ;transmitter busy?
        BPL     RD6             ;yes, loop
        MOVB    LFEED,@#CTXBUF  ;send a line feed
RD7:    TSTB    @#CTXCSR        ;transmitter busy?
        BPL     RD7             ;yes, loop
        MOVB    NULL,@#CTXBUF   ;no, send null, TTY to recover
        TST     (R5)+           ;point spec. buffer for count
        MOV     R1,@(R5)+       ;store # of chars typed in
        MOV     (SP)+,R3        ;restore caller's registers
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        RTS     R5
;
;
;
;       WRITE Sub-routine
;
;
; This sub-routine writes data on the console terminal.
; The address of the output buffer, the address of the
; buffer length argument, and the address of the byte
; count buffer are passed as parameters. If an ASCII null
; character is encountered in the output stream, a carriage
; return and a line feed are generated at the console
; terminal amd control is returned to the caller. The
; number of bytes actually read from the output buffer
```

```
;   is returned in the parameter list.
;
;   Calling Sequence:
;         JSR      R5,WRITE
;         .WORD    OUTBUF      address of output buffer
;         .WORD    LENBUF      address of length argument
;         .WORD    CNTBUF      address of byte count buffer
;   Return Conditions:
;         Number of bytes used from the output buffer is
;         stored in COUNT.
;   Symbolic References:
;         CTXCSR, CTXBUF, CRET, LFEED.
;   Register Usage:
;                  R0 - byte address
;                  R1 - length count
;
WRITE:    MOV      R0,-(SP)          ;save caller's registers
          MOV      R1,-(SP)
          MOV      R2,-(SP)
          MOV      (R5)+,R0          ;get buffer address in R0
          CLR      R1                ;zap length count
WRT1:     CMP      R1,@(R5)          ;entire buffer printed?
          BGE      WRT5              ;yes, return
          INC      R1                ;no, increment length count
          TSTB     (R0)              ;null character?
          BNE      WRT4              ;no, process normally
WRT2:     TSTB     @#CTXCSR          ;transmitter busy?
          BPL      WRT2              ;yes, loop
          MOVB     CRET,@#CTXBUF     ;no, send a carriage return
WRT3:     TSTB     @#CTXCSR          ;transmitter busy?
          BPL      WRT3              ;yes, loop
          MOVB     LFEED,@#CTXBUF    ;no, send a line feed
WRTNUL:   TSTB     @#CTXCSR          ;transmitter busy?
          BPL      WRTNUL            ;yes, loop
          MOVB     NULL,@#CTXBUF     ;no, send null,TTY to recover
          BR       WRT5              ;return
WRT4:     TSTB     @#CTXCSR          ;transmitter busy?
          BPL      WRT4              ;yes, loop
          MOVB     (R0)+,@#CTXBUF    ;no, send a char from buffer
          BR       WRT1              ;go for next character
WRT5:     TST      (R5)+             ;point to count buffer
          MOV      R1,@(R5)+         ;store number of chars written
          MOV      (SP)+,R2          ;restore caller's registers
          MOV      (SP)+,R1
          MOV      (SP)+,R0
          RTS      R5
;
;
;
;         PUT-tAB Sub-routine
;
PUTAB:    MOV      R1,-(SP)          ;save caller's R1
          MOV      #12,R1            ;set loop count
TAB1:     JSR      R5,PUTSPC         ;call ten times
```

app E

```
        DEC     R1
        BNE     TAB1
        MOV     (SP)+,R1            ;restore caller's R1
        RTS     R5
;
;
;
;       PUT-SPaCe Sub-routine
;
PUTSPC: JSR     R5,WRITE            ;write a space
        .WORD   SPACE
        .WORD   BIN001
        .WORD   IOBUFC
        RTS     R5
;
;
;
;       PUT-QuOTe Sub-routine
;
PUTQOT: JSR     R5,WRITE            ;write a double quote
        .WORD   DBLQOT
        .WORD   BIN001
        .WORD   IOBUFC
        RTS     R5
;
;
;
;       PUT-SKiP Sub-routine
;
PUTSKP: JSR     R5,WRITE            ;generate carriage return and
        .WORD   NULL                ;linefeed via writing a null
        .WORD   BIN001
        .WORD   IOBUFC
        RTS     R5
;
;
;
;       PUT-VERsion Sub-routine
;
;  This subroutine fetches the version number and subversion
;  numbers of the ICMSupervisor and prints it in the
;  header.
;
PUTVER: CLR     @#ICMCSR            ;write a 0 for control word
        JSR     R5,GETICM           ;transfer data from ICM
        .WORD   ICMDTA
        MOV     ICMDTA,NUMBFR       ;copy fetched data
        SWAB    NUMBFR              ;align version number to low byte
        BIC     #177400,NUMBFR      ;clear high byte
        JSR     R5,PUTNUM           ;print version number
        JSR     R5,WRITE            ;print a decimal point
        .WORD   POINT
        .WORD   BIN001
        .WORD   IOBUFC
```

```
        MOV     ICMDTA,NUMBFR   ;copy fetched data once more
        BIC     #177400,NUMBFR  ;clear high byte
        JSR     R5,PUTNUM       ;print subversion number
        RTS     R5
;
;
;
;       PUT-EQUal Sub-routine
;
PUTEQU: JSR     R5,WRITE        ;write MSGEQU
        .WORD   MSGEQU
        .WORD   BIN004
        .WORD   IOBUFC
        RTS     R5
;
;
;
;       OCtal-2-BiNary Sub-routine
;
OC2BN:  MOV     R0,-(SP)        ;save caller's registers
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     (R5)+,R2        ;get address of buffer in R2
        CLR     R3
        CLR     @(R5)           ;clear output buffer initially
        ADD     #6,R2           ;6 digits number only valid
OC2BN1: MOVB    -(R2),R0        ;fetch digit
        SUB     #60,R0          ;convert to value
        BMI     OC2BN4          ;not valid char, signal error
        CMP     #10,R0          ;check validity again
        BLE     OC2BN4          ;not valid, signal error
        MOV     R3,R1           ;R3 points to digit position
        MULRX3  R1              ;compute number of shifts
OC2BN2: TST     R1              ;check if done
        BEQ     OC2BN3          ;yes, proceed
        ASL     R0              ;no, shift left one bit
        DEC     R1              ;update # of shifts to be done
        BR      OC2BN2          ;loop till done
OC2BN3: BIS     R0,@(R5)        ;save partial results
        INC     R3              ;update digit pointer
        CMP     #6,R3           ;check if done
        BGT     OC2BN1          ;no, process another digit
        BR      OC2BN5          ;yes, exit
OC2BN4: JSR     R5,WRITE        ;write error message
        .WORD   MSBADC
        .WORD   BIN020
        .WORD   IOBUFC
OC2BN5: TST     (R5)+           ;skip argument list
        MOV     (SP)+,R3        ;restore caller's registers
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        RTS     R5
```

app-E

```
;
;
;
;       BiNary-2-OCtal Sub-routine
;
BN2OC:  MOV     R0,-(SP)        ;save caller's registers
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     (R5)+,R2
        CLR     R3              ;R3-pointer to digit position
        ADD     #6,R2           ;offset pointer for last digit
BN2OC1: MOV     @(R5),R0        ;fetch input number
        MOV     #7,R4           ;load a 3-bit mask
        MOV     R3,R1           ;R1 will contain # of shifts
        MULRX3  R1              ;multiply R1 by 3
BN2OC2: TST     R1              ;check if done
        BEQ     BN2OC3          ;yes, proceed to next step
        ASL     R4              ;no, shift mask one bit left
        DEC     R1              ;update # of shifts to be done
        BR      BN2OC2          ;loop till done
BN2OC3: MOV     R3,R1           ;init R1 again with # of shifts
        MULRX3  R1              ;multiply R1 by 3
        COM     R4              ;invert mask
        BIC     R4,R0           ;mask other bits
BN2OC4: TST     R1              ;check if done
        BEQ     BN2OC5          ;yes, proceed to next step
        ASR     R0              ;no, shift masked bits 1 right
        DEC     R1              ;update # of shifts to be done
        BR      BN2OC4          ;loop till done
BN2OC5: ADD     #60,R0          ;convert to ascii value
        MOVB    R0,-(R2)        ;save converted output
        INC     R3              ;update digit pointer
        CMP     #5,R3           ;check if last digit
        BGT     BN2OC1          ;no, loop till done
        BIT     #100000,@(R5)   ;check bit 15
        BEQ     BN2OC6          ;zero, go for putting zero
        MOVB    #61,-(R2)       ;not 0, put 1(only other poss.)
        BR      BN2OC7          ;then exit
BN2OC6: MOVB    #60,-(R2)       ;put 0 for most signif. digit
BN2OC7: TST     (R5)+           ;skip argument list
        MOV     (SP)+,R4        ;restore caller's registers
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        RTS     R5
;
;
;
;       GET-NUMber Sub-routine
;
GETNUM: MOV     R1,-(SP)        ;save caller's registers
```

```
        MOV     R2,-(SP)
        JSR     R5,READ         ;fetch a number from terminal
        .WORD   IOBUF
        .WORD   BIN006
        .WORD   IOBUFC
        MOV     #IOBUF,R2       ;init R2 with buffer address
        MOV     IOBUFC,R1       ;get number of chars typed in
        CMP     #6,R1           ;check if less or equal to 6
        BGE     GETNM1          ;yes, skip next
        MOV     #6,R1           ;no, limit number to 6
GETNM1: CMP     #6,R1           ;check if number of chars is 6
        BEQ     GETNM4          ;yes, bypass padding lead 0's
        ADD     R1,R2           ;offset byte address
        INC     R2
GETNM2: CMP     #IOBUF+1,R2     ;check if done
        BGE     GETNM3          ;yes, proceed to next step
        MOVB    -2(R2),-(R2)    ;no, shift buffer up by one
        BR      GETNM2          ;loop till done
GETNM3: MOVB    #60,-(R2)       ;store a leading 0 in buffer
        INC     R1              ;update byte count
        BR      GETNM1          ;loop till done
GETNM4: JSR     R5,OC2BN        ;convert string to value
        .WORD   IOBUF
        .WORD   NUMBFR
        MOV     (SP)+,R2        ;restore caller's registers
        MOV     (SP)+,R1
        RTS     R5
;
;
;
;       NUMber-DuMP Sub-routine
;
NUMDMP: JSR     R5,BN2OC        ;convert value to string
        .WORD   IOBUF
        .WORD   NUMBFR
        JSR     R5,WRITE        ;write it on console terminal
        .WORD   IOBUF
        .WORD   BIN006
        .WORD   IOBUFC
        RTS     R5
;
;
;
;       PUT-NUMber Sub-routine
;
;       This subroutine converts a number into an octal
;       ascii string by calling subroutine BN2OC and prints
;       it with leading zeros suppressed.
;
PUTNUM: MOV     R1,-(SP)        ;save caller's registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        JSR     R5,BN2OC        ;convert input to ascii string
        .WORD   IOBUF
```

```
        .WORD   NUMBFR
        MOV     #6,R1              ;string length = 6 chars
PNUM1:  MOV     #IOBUF,R2          ;load pointer to first char
        CMPB    #60,(R2)           ;is char a zero?
        BNE     PNUM3              ;no more leading zeros
        DEC     R1                 ;first one a zero
        MOV     #6,R3              ;suppress by scrolling buffer
PNUM2:  MOVB    1(R2),(R2)+        ;scroll buffer by one char
        DEC     R3
        BNE     PNUM2
        CMP     #1,R1              ;check if done
        BLT     PNUM1              ;if not, loop till done
PNUM3:  MOV     R1,NUMBFR          ;pass length of final string
        JSR     R5,WRITE           ;print the final string
        .WORD   IOBUF
        .WORD   NUMBFR
        .WORD   IOBUFC
        MOV     (SP)+,R3           ;restore registers
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        RTS     R5
;
;
;
;       GETICM Sub-routine
;
GETICM: BIT     #100,@#ICMCSR      ;ICM transmitter full?
        BNE     GETICM             ;no, loop
        MOV     @#ICMBUF,@(R5)+    ;yes, transfer a word from ICM
        RTS     R5
;
;
;
;       PUTICM Sub-routine
;
PUTICM: BIT     #200,@#ICMCSR      ;ICM receiver empty?
        BNE     PUTICM             ;no, loop
        MOV     @(R5)+,@#ICMBUF    ;yes, transfer a word to ICM
        RTS     R5
;
;
;
;       TRap-HANDler Interrupt Service Routine
;
TRHAND: CMP     #400,SP            ;check stack overflow
        BLT     TRH1               ;no, bus time-out occured
        JSR     R5,WRITE           ;yes, write stk o/flow message
        .WORD   MSTKOV
        .WORD   BIN080
        .WORD   IOBUFC
        HALT                       ;fatal error, hang up
TRH1:   JSR     R5,WRITE           ;write bus time-out message
        .WORD   MBUSTO
        .WORD   BIN032
```

```
        .WORD   IOBUFC
        MOV     (SP)+,NUMBFR        ;get return address
        JSR     R5,NUMDMP           ;write it on console terminal
        JSR     R5,PUTSKP
        MOV     #PROCMD,-(SP)       ;push start address
        RTI
        .PAGE
;
;
;
;       CoMmanD-LiST
;
        .EVEN
CMDLST: .ASCII  /HELP/              ;list of commands
        .WORD   HELP
        .ASCII  /GET /
        .WORD   GET
        .ASCII  /PUT /
        .WORD   PUT
        .ASCII  /EDMP/
        .WORD   EDMP
        .ASCII  /ERST/
        .WORD   ERST
        .ASCII  /RUN /
        .WORD   RUN
        .ASCII  /HOLD/
        .WORD   HOLD
        .ASCII  /SDMP/
        .WORD   SDMP
        .ASCII  /INIT/
        .WORD   INIT
        .ASCII  /DIG0/
        .WORD   DIG0
        .ASCII  /DIG1/
        .WORD   DIG1
        .ASCII  /DIG2/
        .WORD   DIG2
        .ASCII  /DIG3/
        .WORD   DIG3
        .ASCII  /DIG4/
        .WORD   DIG4
        .ASCII  /DIG5/
        .WORD   DIG5
        .ASCII  /DIG6/
        .WORD   DIG6
        .ASCII  /DIG7/
        .WORD   DIG7
        .ASCII  /ODMP/
        .WORD   ODMP
        .ASCII  /LOAD/
        .WORD   LOAD
        .ASCII  /CMPR/
        .WORD   CMPR
        .ASCII  /LDPT/
```

app-E

```
        .WORD   LDPT
        .ASCII  /EXIT/
        .WORD   EXIT
        .ASCIZ  //
        .PAGE
;
;
;
;   ASCII strings to be output to Console Terminal
;
        .EVEN
MSTKOV: .ASCIZ  /SYSTEM STACK OVERFLOW, FATAL ERROR./
MBUSTO: .ASCII  /UNIBUS TIME-OUT OCCURRED AT PC= /
MSHDR1: .ASCIZ  /INK  CORRECTION  MONITOR  - V1.1/
MSHDR2: .ASCII  /ICM  SUPERVISOR  - V/                          ;18
MSGCMD: .ASCII  /COMMAND/                                       ;7
MSGEQU: .ASCII  / := /                                          ;4
MSGHLP: .ASCIZ  /USE FOLLOWING COMMANDS ONLY;/
MSGNC1: .ASCIZ  / NOT A VALID COMMAND./
MSGNC2: .ASCIZ  /TYPE "HELP" FOR A LIST OF VALID COMMANDS./
MSGTMP: .ASCIZ  / NOT YET IMPLEMENTED./
MERDET: .ASCIZ  / ERRORS DETECTED./
MSADDR: .ASCII  /SOURCE ADDRESS/                                ;14
MDADDR: .ASCII  /DESTINATION ADDRESS/                           ;19
MWDCNT: .ASCII  /WORD COUNT/                                    ;10
MSCONT: .ASCII  /CONTENTS/                                      ;8
MSBADC: .ASCIZ  /BAD CHARACTER/
MSBADV: .ASCIZ  /BAD VALUE/
LFEED:  .ASCII  <012>
CRET:   .ASCII  <015>
SPACE:  .ASCII  <040>
DBLQOT: .ASCII  <042>
POINT:  .ASCII  <056>
NULL:   .ASCII  <000>
        .PAGE
;
;
;
;   PROGRAM VARIABLES AND MEMORY ALLOCATION
;
;   Numeric Constants
;
        .EVEN
BIN001: .WORD   001
BIN004: .WORD   004
BIN006: .WORD   006
BIN007: .WORD   007
BIN008: .WORD   010
BIN010: .WORD   012
BIN014: .WORD   016
BIN019: .WORD   023
BIN020: .WORD   024
BIN032: .WORD   040
BIN080: .WORD   120
```

```
;
;
;   I/O VARIABLES
;
IOBUF:  .BLKB   120
IOBUFC: .BLKB   2
ICMBFC: .BLKB   2
ICMDTA: .BLKB   2
NUMBFR: .BLKB   2
        .PAGE
;
;
;
;       STATUS TABLES
;
        .EVEN
STBL1:  .ASCIZ  /       COLOR MODE = BLACK/
        .=STBL1+40
        .ASCIZ  /       COLOR MODE = YELLOW/
        .=STBL1+100
        .ASCIZ  /       COLOR MODE = CYAN/
        .=STBL1+140
        .ASCIZ  /       COLOR MODE = MAGENTA/
;
        .EVEN
STBL2:  .ASCIZ  /       BLACK COLOR SUPPRESSED/
        .=STBL2+40
        .ASCIZ  /       YELLOW COLOR SUPPRESSD/
        .=STBL2+100
        .ASCIZ  /       CYAN COLOR SUPPRESSED/
        .=STBL2+140
        .ASCIZ  /       MAGENTA COLOR SUPPRESSED/
        .=STBL2+200
        .ASCIZ  /       NO COLOR SUPPRESSED/
;
        .EVEN
STBL3:  .ASCIZ  /       OP-MODE = IDLE/
        .=STBL3+40
        .ASCIZ  /       OP-MODE = TRANSPARENT/
        .=STBL3+100
        .ASCIZ  /       OP-MODE = PROCESS/
        .=STBL3+140
        .ASCIZ  /       OP-MODE = SPARE/
        .PAGE
;
;
;
;       ERROR MESSAGES TABLE
;
        .EVEN
ERRTBL: .ASCIZ  /   NO  ERROR/
        .=ERRTBL+40
        .ASCIZ  /ERROR 1 - RTP NOT INITIALIZED/
        .=ERRTBL+100
```

```
        .ASCIZ  /ERROR 2 - HHKD NON-COMMAND KEY/
        .=ERRTBL+140
        .ASCIZ  /ERROR 3 - HHKD IMPROPER KEY/
        .=ERRTBL+200
        .ASCIZ  /ERROR 4 - MEM WRITE OP FAILED/
        .=ERRTBL+240
        .ASCIZ  /ERROR 5 - HHKD IMPROPER KEY/
        .=ERRTBL+300
        .ASCIZ  /ERROR 6 - EMPTY LOC. ADDRESSED/
        .=ERRTBL+340
        .ASCIZ  /ERROR 7 - RAMTST ARGUMENT BAD/
        .=ERRTBL+400
        .ASCIZ  /ERROR 8 - RUN-MODE SWITCH BAD/
        .=ERRTBL+440
        .ASCIZ  /ERROR 9 - RTP PC OVER-RANGE/
        .=ERRTBL+500
        .ASCIZ  /ERROR 10 - ICM HUNG UP BY HOP/
        .PAGE
;
;
;
;MEMORY ALLOCATION FOR MICROPROGRAMS AND INK-CORRECTION TABLES
;
        .EVEN
MICPGM: .BLKW   4000
INKTBL: .BLKW   3000
;
        .END
```

APPENDIX - F

8085A-BASED ICM RESIDENT SUPERVISOR -- ICMS.8080

This is the Supervisor for Ink Correction Module.
It resides in 2708 EPROMs installed on ICM-Manager board.
Its objective is to provide the following;

1) Initialization of ICM.
2) Handshake-operation with the ICMON (Ink Correction MONitor), which is a Command Interpreter operating on the host PDP-11 computer.
3) A Hex/Command-Keyboard-LED-Display monitor called KHDMON, which offers debugging as well as program development facility in the same style as HAL-monitor operating on NEC's TK-80A microcomputer.
4) A collection of routines for controlled operation of the ICM Real-Time Processor.
5) A collection of RTP diagnostic routines.

The program is activated through power-on reset or manually from RESET push-buttons, following which, initialization routine is executed and then the control is transferred to KHDMON. The control always remains with KHDMON unless an interrupt occurs, whereupon, appropriate service routine is entered. Control is eventually returned to KHDMON after the interrupt has been serviced.

Interrupts are used for the following purposes;

INTR - trap for non-existent memory addressing
RST5.5 - single step facility of KHDMON
RST6.5 - handshake-operation with ICMON
RST7.5 - monitor ICM status changes
TRAP - watchdog for UNIBUS hang-up situations

ERROR Handling : In case of error condition, error LED is turned on and a error-code number is stored in a reserved location in the control block. This code can be viewed by activating ErrorDuMP routine. Error condition is reset by activating ErrorRESET routine. Errors are qualified according to the following list;

01 - System not initialized after RESET/Status-change.
02 - Non-command key while Supervisor wants command key.
03 - Improper key for register pair display.
04 - Memory 'write' not successful.

```
;      05 - Improper key for register display.
;      06 - Non-existent memory addressed.
;      07 - RAMTST's memory boundary violation.
;      08 - RUN-Mode switch contents not valid.
;      09 - RTP Program Counter over-range.
;      0A - ICM hung up by host processor.
;
;
;         EDIT HISTORY :
;
;JUL-23-80       V1.0    SNM     Original .
;JAN-10-81       V1.1    SNM     Diagnostic Routines added.
;
;
;
;
;   User I/O via PDP-11 Interface
;   Debugging thru HHKD (Hand-Held Keyboard/Display module)
;
;
;        INK  CORRECTION  MODULE  SUPERVISOR
;
;
;   GENERAL EQUATES
;
VER      =        /01      ;ICMSUP Version Number
SUBVER   =        /01      ;ICMSUP Sub-Version Number
SIM      =        /30      ;SIM instr not implem. on mical
RIM      =        /20      ;RIM instr not implem. on mical
;
;
;   I/O  EQUATES
;
DA1HOS   =        /A0      ;Host Interface Data Port A1
DB1HOS   =        /A1      ;Host Interface Data Port B1
DC1HOS   =        /A2      ;Host Interface Data Port C1
C1HOST   =        /A3      ;Host Interface Control Port 1
DA2HOS   =        /B0      ;Host Interface Data Port A2
DB2HOS   =        /B1      ;Host Interface Data Port B2
DC2HOS   =        /B2      ;Host Interface Data Port C2
C2HOST   =        /B3      ;Host Interface Control Port 2
DA1RTP   =        /80      ;RTP Interface Data Port A1
DB1RTP   =        /81      ;RTP Interface Data Port B1
DC1RTP   =        /82      ;RTP Interface Data Port C1
C1RTP    =        /83      ;RTP Interface Control Port 1
DA2RTP   =        /90      ;RTP Interface Data Port A2
DB2RTP   =        /91      ;RTP Interface Data Port B2
DC2RTP   =        /92      ;RTP Interface Data Port C2
C2RTP    =        /93      ;RTP Interface Control Port 2
D8279    =        /00      ;8279 Data Port
C8279    =        /01      ;8279 Control Port
TRACK    =        /C0      ;TRAP acknowledge
ARMST5   =        /D0      ;rearm single-step thru RST5.5
WAITIN   =        /E0      ;pause for HOST response(drop)
```

```
WAITOT  =       /F0     ¦pause for HOST response(pick)
ACOBI   =       /82     ¦port A, C out B in
ABICO   =       /92     ¦port A, B in C out
AM2BI   =       /FF     ¦port A mode 2, B mode 1 in
CLKMDL  =       /08     ¦RTP clock turn-off template
CLKMDH  =       /09     ¦RTP clock turn-on template
IGNDL   =       /09     ¦IGNORDTA' turn-off template
IGNDH   =       /08     ¦IGNORDTA' turn-on template
ICMBSY  =       /0E     ¦ICMRDY turn-off template
ICMRDY  =       /0F     ¦ICMRDY turn-on template
DHOLD   =       /0A     ¦DATAHOLD' turn-on template
DFLOW   =       /0B     ¦DATAHOLD' turn-off template
DSTEPL  =       /0D     ¦DATASTEP' turn-off template
DSTEPH  =       /0C     ¦DATASTEP' turn-on template
RRUNL   =       /00     ¦RTPRUN turn-off template
RRUNH   =       /01     ¦RTPRUN turn-on template
ERRL    =       /02     ¦ERROR turn-off template
ERRH    =       /03     ¦ERROR turn-on template
SCLRL   =       /04     ¦SYSCLR turn-off template
SCLRH   =       /05     ¦SYSCLR turn-on template
RINTL   =       /07     ¦RTPINT' turn-off template
RINTH   =       /06     ¦RTPINT' turn-on template
CINL    =       /0A     ¦CIN turn-off template
CINH    =       /0B     ¦CIN turn-on template
RLDL    =       /0D     ¦RLD turn-on template
RLDH    =       /0C     ¦RLD turn-off template
RJMPL   =       /0E     ¦RJMP turn-off template
RJMPH   =       /0F     ¦RJMP turn-on template
STMSK   =       /0F     ¦CDF status mask
SLPMSK  =       /10     ¦RTPSLP mask
NWDMSK  =       /20     ¦NEWDTA mask
STFMSK  =       /40     ¦STKFUL mask
ACKMSK  =       /80     ¦RTPACK mask
VEC8L   =       /00     ¦VECT8 turn-off template
VEC8H   =       /01     ¦VECT8 turn-on template
VEC9L   =       /02     ¦VECT9 turn-off template
VEC9H   =       /03     ¦VECT9 turn-on template
VEC10L  =       /04     ¦VECT10 turn-off template
VEC10H  =       /05     ¦VECT10 turn-on template
VEC11L  =       /06     ¦VECT11 turn-off template
VEC11H  =       /07     ¦VECT11 turn-on template
FIFO    =       /40     ¦8279 command code to read FIFO
IOMODE  =       /00     ¦8279 command code to set I/O
                        ¦  mode for 8 8-bit character
                        ¦  display -left entry and
                        ¦  Encoded scan keyboard
                        ¦             - 2 key lockout
¦
¦
¦  PROGRAM EQUATES
¦
BASE    =       /0000   ¦Supervisor origin
MCROPC  =       /7000   ¦RTP program counter address
NEXT    =       /15     ¦value of NEXT key
```

```
CLEAR   =       /17      ¦value of CLEAR key
LEDASH  =       /40      ¦7-segment LED DASH
RAMLEN  =       /08      ¦length of display RAM
LEDE    =       /79      ¦7-segment LED capital E
LEDR    =       /50      ¦7-segment LED low-case r
LEDG    =       /3D      ¦7-segment LED capital G
LEDO    =       /5C      ¦7-segment LED low-case o
LEDD    =       /5E      ¦7-segment LED low-case d
LEDB    =       /7C      ¦7-segment LED low-case b
LEDP    =       /F3      ¦7-segment LED cap P with a pt.
PCLK    =       /34      ¦8279 clock init template
¦
¦
¦
¦
¦  Execution starts here after RESET
¦
START:  .CSECT           ¦start assembly
        .=BASE           ¦address origin
        JMP     INIT     ¦branch to Init routine
¦
¦
        .=/08            ¦address restart 1
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
        .=/10            ¦address restart 2
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
        .=/18            ¦address restart 3
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
        .=/20            ¦address restart 4
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
        .=/24            ¦address TRAP vector
        JMP     TRHAND   ¦branch to TRap HANDler
¦
¦
        .=/28            ¦address restart 5
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
        .=/2C            ¦address RST5.5 vector
        JMP     R5HAND   ¦branch to Rst5.5 HANDler
¦
¦
        .=/30            ¦address restart 6
        JMP     INTHND   ¦branch to interrupt handler
¦
¦
```

```
        .=/34                   ;address RST6.5 vector
        JMP      R6HAND         ;branch to Rst6.5 HANDler
;
;
        .=/38                   ;address restart 7
        JMP      INTHND         ;branch to interrupt handler
;
;
        .=/3C                   ;address RST7.5 vector
        JMP      R7HAND         ;branch to Rst7.5 HANDler
;
;
;   INITIALIZATION ROUTINE
;
;       RTP Interface Initialization
;
INIT:   OUT      TRACK          ;init watchdog on TRAP input
        MVI      A,ACOBI        ;set RTP control port 1
        OUT      C1RTP          ;Init mode command
        MVI      A,ABICO        ;set RTP control port 2
        OUT      C2RTP          ;Init mode command
        MVI      A,/6C          ;load initial control byte
        OUT      DC1RTP         ;Init port C1 on RTP Interface
        MVI      A,/70          ;load initial control byte
        OUT      DC2RTP         ;Init port C2 on RTP Interface
;
;       HOST Interface Initialization
;
        MVI      A,AM2BI        ;set HOST Interface cont. port
        OUT      C1HOST         ;Init mode command
        OUT      C2HOST         ;Init mode command
;
;       8279 Initializations
;
;       IOMODE programs the 8279's keyboard/display mode.
;       FIFO sets the 8279 data reads for the FIFO RAM.
;       All data reads are assumed to be from the FIFO RAM.
;       If subsequent changes take place, the FIFO command
;         must be moved to the key sub-routine.
;
        MVI      A,IOMODE       ;set 8279 I/O mode
        OUT      C8279          ;init 8279 command port
        MVI      A,FIFO         ;set 8279 data reads for FIFO
        OUT      C8279          ;init 8279 command port
        MVI      A,PCLK         ;set 8279 clock devider
        OUT      C8279          ;init 8279 clock
;
;       Control Block & System Stack Initialization
;
        LXI      H,BRKA         ;address next loc. in CBLOCK
        XRA      A              ;to flush the rest of CBLOCK
INIT1:  MOV      M,A            ;clear location
        INR      L              ;point to next location
        JNZ      INIT1          ;loop until done
```

```
KHDMON: LXI     SP,STACK¦purge system stack
        PUSH    H       ¦save user HL at SAVHL
        PUSH    B       ¦save user BC at SAVBC
        PUSH    D       ¦save user DE at SAVDE
        PUSH    PSW     ¦save user AF at SAVAF
        LXI     H,/1000 ¦HL <- starting RAM address
        SHLD    RAMPTR  ¦init user RAM pointer
        SHLD    SAVPC   ¦init user program counter
        MVI     A,/19   ¦load RST mask init template
        SIM             ¦init RST mask
        CALL    R7HAND  ¦get handler to init status
GOCMD:  LXI     H,0     ¦clear HL
        DAD     SP      ¦HL <- value of stack pointer
        SHLD    SAVSP   ¦save user stack pointer
        CALL    CL47    ¦clear LEDs 4 thru 7(right)
        CALL    DLPC    ¦display (on LEDs) user-PC
        LDA     DSPLM   ¦get display mode cont. switch
        ORA     A       ¦test if zero
        JNZ     CRG1    ¦no, goto reg display mode
        CALL    DLHLP   ¦yes, display HL-pointed value
        JMP     GETCOM  ¦get a command from keyboard
¦
¦
¦
¦   Restart 7 INTeRrupt HaNDler routine
¦
¦       Activated by hardware INTR signal as a result of
¦       non-existent memory addressing or Restart 7 (FF) instr.
¦
INTHND: DI              ¦disable intr for RST7 entry
        XTHL            ¦HL <- PC, orig HL on stack
        SHLD    PANBOX  ¦save PC for later analysis
        PUSH    D       ¦save user's DE
        MVI     E,/06   ¦load error code
        CALL    ERROR   ¦display error
        POP     D       ¦restore user's DE
        LXI     H,GETCOM¦prepare to exit thru GETCOM
        XTHL            ¦rest. HL,addr GETCOM on stack
        EI              ¦enable interrupts
        RET
¦
¦
¦
¦   ReSTart 5.5 Interrupt Handler routine
¦
¦   Activated by hardware RST5.5 signal, generated by
¦   single-step circuit. Triggered by M1-states, the
¦   signal goes high after execution of 3 instructions
¦   from reset. The circuit is reset by activation of
¦   ARMST5 signal.
¦
R5HAND: XTHL            ¦HL <- UserPC, org. HL on stack
        SHLD    SAVPC   ¦save UserPC in CBLOCK
        PUSH    B       ¦save BC on stack
```

```
        PUSH    D       ;save DE on stack
        PUSH    PSW     ;save AF on stack
        RIM             ;fetch current mask
        ANI     /07     ;blank non-relevant bits
        ORI     /09     ;modify mask
        SIM             ;set new mask, RST5.5 disabled
        EI              ;enable rest of the interrupts
        LDA     RUNM    ;get Run Mode control switch
        ORA     A       ;check if 0, Mode=Freerun
        JZ      BRKRUN  ;yes, then go check breakpoint
        DCR     A       ;check for single-step mode
        JZ      GOCMD   ;if so, go process a command
        MVI     E,/08   ;invalid RUNM, load error code
        CALL    ERROR   ;signal error condition
        JMP     USERPC  ;and exit
;
;
;
;
;       Check BREAKPOINT after RST5.5 interrupts RUN MODE
;
BRKRUN: XCHG            ;DE <- user PC
        LHLD    BRKA    ;get breakpoint address
        MOV     A,E     ;get LSB of user PC
        CMP     L       ;compare with LSB of BRKA
        JNZ     USERPC  ;different, return to user prog
        MOV     A,D     ;same, now get MSB of user PC
        SUB     H       ;subtract MSB of BRKA
        JNZ     USERPC  ;different, return to user prog
        LXI     H,BRKD  ;same,point HL to BRKD
        ORA     M       ;get BRKD in ACC and set flags
        JZ      USERPC  ;no depth, return to user prog
        DCR     A       ;if depth, count it down
        JZ      GOCMD   ;zero, process a command
        MOV     M,A     ;non-zero, store new depth
        JMP     USERPC  ;return to user program
;
;
        .=/0100         ;align page boundary for tables
;
;       COMMAND VECTOR TABLE
;
;       Offset into this table by command key value
;       Table entry contains complete address of
;       command process routine.
;
;         This table must not cross pages.
;
COMTBL: .WORD   CMEM    ;MEM key,   value = /10
        .WORD   CREG    ;REG key,   value = /11
        .WORD   CADDR   ;ADDR key,  value = /12
        .WORD   CSTEP   ;STEP key,  value = /13
        .WORD   CRUN    ;RUN key,   value = /14
        .WORD   CNEXT   ;NEXT key,  value = /15
        .WORD   CBKPT   ;BKPT key,  value = /16
```

```
        .WORD   GETCOM  ¦CLEAR key, value = /17
        .WORD   CERST   ¦ERST key,  value = /18
        .WORD   CMTEST  ¦RAMTST key,value = /19
        .WORD   CRPCLR  ¦RPCLR key, value = /1A
        .WORD   CVECT   ¦VECTOR key,value = /1B
        .WORD   CRPRUN  ¦RPRUN key, value = /1C
        .WORD   CRPBRK  ¦RPBRK key, value = /1D
        .WORD   CRPSIN  ¦RPSING key,value = /1E
        .WORD   CRPHLD  ¦RPHOLD key,value = /1F
        .WORD   CDIG0   ¦DIG0 key,  value = /20
        .WORD   CDIG1   ¦DIG1 key,  value = /21
        .WORD   CDIG2   ¦DIG2 key,  value = /22
        .WORD   CDIG3   ¦DIG3 key,  value = /23
        .WORD   CDIG4   ¦DIG4 key,  value = /24
        .WORD   CDIG5   ¦DIG5 key,  value = /25
        .WORD   CDIG6   ¦DIG6 key,  value = /26
        .WORD   CDIG7   ¦DIG7 key,  value = /27
¦
¦
¦
¦       HEX DISPLAY TABLE
¦
¦       Hex-digit to 7-segment LED conversion
¦
¦               Offset into this table by hex key value
¦               One-byte table entry is 7-segment LED code
¦               This table must not cross pages.
¦
HEXTBL: .BYTE   /3F     ¦0
        .BYTE   /06     ¦1
        .BYTE   /5B     ¦2
        .BYTE   /4F     ¦3
        .BYTE   /66     ¦4
        .BYTE   /6D     ¦5
        .BYTE   /7D     ¦6
        .BYTE   /07     ¦7
        .BYTE   /7F     ¦8
        .BYTE   /6F     ¦9
        .BYTE   /77     ¦A
        .BYTE   /7C     ¦B
        .BYTE   /39     ¦C
        .BYTE   /5E     ¦D
        .BYTE   /79     ¦E
        .BYTE   /71     ¦F
¦
¦
¦
¦       REGISTER DISPLAY TABLE
¦
¦       Register name display and location in stack
¦
¦       Consists of 8 2-byte entries. Offset into
¦       this table by hex-key value (where 8 is REG H and
¦       9 is REG L). A table entry consists of:
```

```
;               1) A 1-byte 7-segment LED code for the
;                  corresponding register name.
;               2) A 1-byte offset into the stack from the
;                  stored stack pointer. Upon entry to the
;                  monitor, all the processor's registers
;                  are saved on the stack in a given order
;                  which permits DLREG to predict their
;                  location for display or modification.
;
;       This table must not begin in the lower 15 bytes of any
;       page, nor cross pages.
;
REGTBL: .BYTE   /76,7   ;display H, offset 7 bytes
        .BYTE   /38,6   ;display L, offset 6 bytes
        .BYTE   /77,1   ;display A, offseyt 1 byte
        .BYTE   /7C,5   ;display B, offset 5 bytes
        .BYTE   /39,4   ;display C, offset 4 bytes
        .BYTE   /5E,3   ;display D, offset 3 bytes
        .BYTE   /79,2   ;display E, offset 2 bytes
        .BYTE   /71,0   ;display F, no offset
;
;
;
;       REGISTER PAIR TABLE
;
;       Register-pair key value to register-pair name display
;
;       Consists of 5 3-byte table entries. Look-up routine
;       attempts match on first byte of eentry, and must keep
;       track of such iterations to detect errors. A typical
;       table entry contains:
;               1) A hex-key value which represents a
;                  particular register pair request.
;               2) A 2-byte value which is the corresponding
;                  display pattern in 7-segment LED code.
;
;       This table must not cross pages.
;
RGPTBL: .BYTE   /01,/6D,/F3;key#1, display SP.
        .BYTE   /02,/6D,/B1;key#2, display ST.
        .BYTE   /08,/76,/B8;key#8, display HL.
        .BYTE   /0B,/7C,/B9;key#B, display BC.
        .BYTE   /0D,/5E,/F9;key#D, display DE.
;
;
;
;  ERROR sub-routine turns 'ERROR' LED ON
;       Displays 'Error' and code on LED display.
;       Expects error code in register E.
;
ERROR:  PUSH    H       ;save caller's HL
        PUSH    D       ;save caller's DE
        PUSH    B       ;save caller's BC
        PUSH    PSW     ;save caller's AF
```

```
        CALL    CL07    ¦clear all LEDs and A, B
        MOV     A,E     ¦copy error code in ACC
        STA     ERCODE  ¦save error code in CBLOCK
        CALL    DLA     ¦display error code
        MVI     A,ERRH  ¦set ERROR turn-on template
        OUT     C1RTP   ¦turn LED on
        MVI     H,0     ¦point to LED#0
        MVI     L,LEDE  ¦load capital E for display
        CALL    DSPLAY  ¦display on LEDs
        INR     H       ¦point ot next LED
        MVI     L,LEDR  ¦load lo-case r for display
        CALL    DSPLAY  ¦display on LEDs
        INR     H       ¦point to next LED
        CALL    DSPLAY  ¦display another lo-case r
        INR     H       ¦point to next LED
        MVI     L,LEDO  ¦load lo-case o for display
        CALL    DSPLAY  ¦display on LEDs
        INR     H       ¦point to next display
        MVI     L,LEDR  ¦load lo-case r for display
        CALL    DSPLAY  ¦display on LEDs
        POP     PSW     ¦restore caller's AF
        POP     B       ¦restore caller's BC
        POP     D       ¦restore caller's DE
        POP     H       ¦restore caller's HL
        RET
¦
¦
¦
¦  ERESET sub-routine clears ERCODE in CBLOCK, turns LED off
¦       no parameters are passed either way
¦
ERESET: PUSH    H       ¦save caller's HL
        PUSH    D       ¦save caller's DE
        PUSH    B       ¦save caller's BC
        PUSH    PSW     ¦save caller's AF
        XRA     A       ¦clear A
        STA     ERCODE  ¦clear ERCODE in CBLOCK
        CALL    DLA     ¦display 0 now as error code
        MVI     A,ERRL  ¦set ERROR turn-off template
        OUT     C1RTP   ¦turn LED off
        POP     PSW     ¦restore caller's AF
        POP     B       ¦restore caller's BC
        POP     D       ¦restore caller's DE
        POP     H       ¦restore caller's HL
        RET
¦
¦
¦
¦       NBBL--Convert Accumulator to Two Nibbles
¦               Expects value in A
¦               Returns low nibble in A, high nibble in B
¦
NBBL:   MOV     C,A     ¦save original value
        ANI     /F0     ¦keep only upper nibble
```

```
        RLC                     ;switch into lower nibble
        RLC
        RLC
        RLC
        MOV     B,A             ;put into lower nibble of B
        MOV     A,C             ;restore original value
        ANI     /0F             ;keep only lower nibble
        RET
;
;
;
;       GET A HEXKEY AND SHIFT IT INTO LOWER NIBBLE OF HL
;
;               Uses B
;               Returns shifted HL
;
;               On command key-termination, exits with a
;               dummy pop followed by a RET.  This effects
;               a return to the caller of the routine
;               which called GETH.
;
GETH:   CALL    KEY             ;get a key
        JC      GH1             ;if hex key, continue
        MOV     B,A             ;if comm. key, propagate value
        POP     D               ;dummy pop to abort normal ret
        POP     D               ;restore key count in D
        RET                     ;return to caller of caller
GH1:    DAD     H               ;shift HL left four positions
        DAD     H
        DAD     H
        DAD     H
        ORA     L               ;OR with latest key value
        MOV     L,A
        RET                     ;normal return to caller
;
;
;
;       GET WORD (TWO BYTES) FROM HEXKEYS
;
;               Uses B,C,E
;               Returns key count in D
;               Returns terminating command key in A and B
;               Returns word value in HL
;
;               Displays word value in leds #0-#3
;
GETW:   LXI     H,0             ;initialize word value
        PUSH    H               ;initialize key count on stack
GW1:    CALL    GETH            ;get a hex key
        CALL    DLHL            ;display value in progress
        POP     D               ;restore key count from stack
        INR     D               ;increment key count
        PUSH    D               ;put it back on stack
        CALL    DLHLP           ;display pointed byte
```

```
        JMP     GW1      ;do again until GETH exists
;
;
;
;       GET BYTE FROM HEX KEYS
;
;               Uses B,C,E
;               Returns key count in D
;               Returns terminating command key in A and B
;               Returns byte value in HL
;
;               Displays byte value in leds #6-#7
;
GETB:   LXI     H,0      ;initialize byte value
        PUSH    H        ;initialize key count on stack
GB1:    CALL    GETH     ;get a hex key
        MOV     A,L      ;prepare byte for display
        CALL    DLA      ;display it in rightmost leds
        POP     D        ;restore key count from stack
        INR     D        ;increment key count
        PUSH    D        ;put it back on stack
        JMP     GB1      ;do again until GETH exists
;
;
;
;       GETCOM GETS A COMMAND KEY AND PROCESSES IT
;               Gets control after initialization and upon
;               recognized exit from some commands--other
;               commands may unpredictably acquire a command
;               key, then enter GETCOM at optional entry
;               points GC1 and GC2.
;
GETCOM: CALL    KEY      ;get a hex key command in A & C
        JNC     GC0      ;if comm key, bypass error call
        MVI     E,/02    ;set error code
        CALL    ERROR    ;and call ERROR routine
        JMP     GETCOM   ;on return, loop for proper key
GC0:    LXI     H,ERCODE ;point to ERCODE in CBLOCK
        MOV     E,M      ;fetch error code
        DCR     E
        DCR     E        ;check if last error due to key
        CZ      ERESET   ;yes, reset error, now comm key
        LHLD    SAVPC    ;prep HL with user PC for later
GC1:    MVI     D,0      ;init hex key counter
GC2:    PUSH    H        ;save user PC on stack for now
        SUI     /10      ;subtr offset 16 from key value
        ADD     A        ;2X key value to obtain offset
        MOV     C,A      ;copy offset into C
        CALL    CL47     ;clear leds #4-#7
        LXI     H,COMTBL ;point to command table
        XRA     A        ;clear ACC for command routines
        MOV     B,A      ;zero MSB of register pair BC
        DAD     B        ;offset pointer to comm addr LSB
        MOV     C,M      ;set up vector LSB in reg C
```

```
        INX     H           ¦point to next loc to get MSB
        MOV     H,M         ¦set up vector MSB
        MOV     L,C         ¦HL= addr of comm proc routine
        LXI     B,DSPLM     ¦set up BC for command routines
        XTHL                ¦HL=addr parm, comm addr on stack
        RET                 ¦go to selected command routine
¦
¦
¦
¦
¦       REGISTER DISPLAY SUBROUTINE
¦
¦       DLREG displays a register-name legend and a dash
¦       followed by the value of that register at entry
¦       to the monitor (after reset, step, or breakpoint).
¦
¦       Uses A,B,D,E
¦       Expects stopped register key value in DSPLM
¦         (display mode).  If zero, this routine
¦         will invoke default HL-pointed format.
¦
DLREG:  LDA     DSPLM       ¦get display mode indicator
        ADD     A           ¦double it
        JZ      DLHLP       ¦if zero, use HL-pointed mode
        LXI     H,REGTBL-/10¦point to register table
        ADD     L           ¦add register key displacement
        MOV     L,A
        PUSH    H           ¦save HL on stack for now
        MOV     L,M         ¦get register-name display in L
        MVI     H,/04       ¦point to LED#4
        CALL    DSPLAY      ¦put reg-name in display led #4
        MVI     L,LEDASH    ¦get a dash in 7-segment code
        INR     H           ¦point to next LED
        CALL    DSPLAY      ¦put dash in led #5
        POP     H           ¦restore HL
        INX     H           ¦point to 2nd byte REGTBL entry
        MOV     E,M         ¦get the stack displacement
        MVI     D,0         ¦zero msb of DE register pair
        LHLD    SAVSP       ¦get stored stack pointer
        DAD     D           ¦add the displacement
        JMP     DLHLP       ¦display the saved reg value
¦
¦
¦
¦
¦       KEYBOARD sub-routine
¦
¦       Reads the 8279 FIFO until a valid key is depressed.
¦       Uses B and D.
¦       Returns key value in A and C.
¦       Returns carry set if a hex-key was depressed.
¦       Returns carry not set if a command-key was depressed.
¦
¦       Key value returned is /00-/0F for hex-keys or /10-/27
¦       for command keys. Format is:
¦                                       000C CRRR
```

```
;               where CC = 00 for keys 0-7
;                          01 for keys 8-F
;                          10 for command-keys
;                     RRR = binary value of row 0-7
;
;
KEY:    IN      C8279   ;read FIFO status word
        ANI     /07     ;set zero if FIFO is empty
        JZ      KEY     ;loop until a key is depressed
        IN      D8279   ;read FIFO
        MOV     B,A     ;move A to register B
        ANI     /07     ;obtain return bits
        RLC             ;shift return bits left
        RLC
        RLC
        MOV     D,A     ;move A to D
        MOV     A,B     ;move FIFO word to A
        ANI     /38     ;obtain scan bits
        RRC             ;shift scan bits right
        RRC
        RRC
        ADD     D       ;return + scan bits = key value
        CPI     /28     ;set carry if value < /28
        JNC     KEY     ;if not set, loop for valid key
        CPI     /10     ;set carry if value</10, hex-key
        MOV     C,A     ;copy A into C
        RET
;
;
;
;       CLEAR-DISPLAYS subroutines
;
;               CL07 clears display leds #0-#7
;               CL47 clears display leds #4-#7
;
;               Uses H,L,B and A
;
CL07:   MVI     B,/08   ;set to clear 8 leds
        JMP     CL      ;begin clear
CL47:   MVI     B,/04   ;set to clear 4 leds
CL:     MVI     L,0     ;set clearing value
        MVI     H,/07   ;point to rightmost led (#7)
CL1:    CALL    DSPLAY  ;clear one led
        DCR     H       ;point to next lower led #
        DCR     B       ;count down loop control
        JNZ     CL1     ;clear again until done
        RET
;
;
;       DSPLAY subroutine displays 1 hex character
;
;               H contains 8279 ram address
;               L contains character to be displayed
;                 the 8279 ram command is
```

```
;                  1000 OAAA where AAA = ram location
;
;               Uses A
;
DSPLAY: MOV     A,H     ;8279-display RAM address to A
        CPI     RAMLEN  ;set carry if address is valid
        RNC             ;return if carry is not set
        ORI     /80     ;set up A with 8279 ram command
        OUT     C8279   ;program 8279 for next RAM loc.
        MOV     A,L     ;move character to A
        OUT     D8279   ;write display char to 8279
        RET
;
;
;
;       BYTE-SIZE display subroutines
;
;
;       DLHLP displays the value pointed to by HL
;       DLA displays the accumulator.
;           both of above display in rightmost leds.
;       DLADE displays A to leds pointed to by D .
;
;               Uses A,B,C,D
;
DLHLP:  MOV     A,M     ;get value to display in ACC
DLA:    MVI     D,/07   ;point to rightmost led (#7)
DLADE:  PUSH    H       ;save caller's HL
        CALL    NBBL    ;divide A into two nibbles
        CALL    DLHEX   ;display first nibble as hex
        MOV     A,B     ;prepare second nibble
        CALL    DLHEX   ;display second nibble as hex
        POP     H       ;restore caller's HL
        RET
;
;
;
;
;       DLHEX - Display a nibble in A as one hex digit
;
;       Expects value in low nibble of A
;       Expects D pointing to LED display number
;       Returns D one smaller
;
;       Uses H,L
;
DLHEX:  LXI     H,HEXTBL;point to conversion table
        ADD     L       ;add in accumulator
        MOV     L,A
        MOV     L,M     ;fetch 7-segment display value
        MOV     H,D     ;mov display address to H
        CALL    DSPLAY  ;display nibble
        DCR     D       ;point D to next lower led
        RET
;
;
;
```

```
;       WORD-SIZE display subroutines
;
;       DLPC displays user's PC (SAVPC) in leds.
;       DLHL displays current HL in leds
;            both of above display at leftmost leds (#0-#3).
;
;       DLHLDE displays current HL in leds pointed to by DE .
;
;       Uses A,B,C,D,E .
;
;               DLPC uses HL also .
;               DLHL and DLHLDE return HL unchanged.
;
DLPC:   LHLD    SAVPC   ;fetch user PC into HL
DLHL:   MVI     D,/03   ;point to left-half LEDs 0-3
DLHLDE: MOV     A,L     ;set up L for display
        CALL    DLADE   ;display it in LEDs 2-3
        MOV     A,H     ;set up H for display
        JMP     DLADE   ;display it in LEDs 0-1 & ret
;
;
;
;
;       COMMAND KEY PROCESSES
;
;
;
;
;
;       COMMAND BREAKPOINT PROCESS
;
;       If no address parameter preceded, displays existing
;         breakpoint address and depth.
;       If address parameter preceded, displays BP. and
;         awaits depth parameter.
;       If CLEAR entered for depth parameter, breakpoint
;         address (BRKA) and depth (BRKD) are cleared.
;       If valid terminator on depth parameter, breakpoint
;         address and depth is set and displayed.
;
CBKPT:  PUSH    H       ;save pending address param
        MVI     H,/04   ;point to LED#4
        MVI     L,LEDB  ;load lo-case b for display
        CALL    DSPLAY  ;display on LEDs
        INR     H       ;point to next LED
        MVI     L,LEDP  ;load capital P. for display
        CALL    DSPLAY  ;display on LEDs
        POP     H       ;restore addr param, if any
        MOV     A,D     ;check for pending addr param
        ORA     A       ;any pending?
        JZ      DLBK    ;none, display present breakpt.
        PUSH    H       ;yes, save it again on stack
        CALL    GETB    ;get a byte from hex keys
        CPI     CLEAR   ;was delimiter a CLEAR key?
        MOV     A,L     ;get entered value in ACC
        POP     H       ;restore address parameter
```

```
        JNZ     SETBP   ¦not CLEAR key, set breakpoint
        LXI     H,0     ¦yes, set up for clearing
        XRA     A
SETBP:  SHLD    BRKA    ¦store the breakpoint address
        STA     BRKD    ¦store the breakpoint depth
        MOV     A,B     ¦check terminator key again
        CPI     CLEAR   ¦was it CLEAR key?
        JNZ     GC1     ¦no, process terminator command
DLBK:   LHLD    BRKA    ¦fetch the breakpoint address
        CALL    DLHL    ¦display it on LED0 thru LED3
        LDA     BRKD    ¦fetch the breakpoint depth
        CALL    DLA     ¦display it on LED6 and LED7
        JMP     GETCOM  ¦go process another command.
¦
¦
¦
¦       COMMAND NEXT PROCESS
¦
¦       Next key encountered by main GETCOM loop
¦       implies termination of an addr parameter.
¦       Note that next key during reg inspection
¦       is handled separately under command reg
¦       process.  Here, routine sets display mode
¦       to HL-pointed format after incrementing
¦       the HL pointer.  Processing continues in
¦       command MEM routine.
¦
CNEXT:  INX     H       ¦increment memory pointer
        STAX    B       ¦zero display mode(HL-pointed)
        JMP     SETPTR  ¦go to set RAM pointer
¦
¦
¦
¦       COMMAND STEP AND RUN PROCESSES
¦
¦       Sets RUN Mode switch for STEP or RUN ( 1 or 0 ).
¦       If no address preceded, assumes pre-existing
¦       user PC in CBLOCK's SAVPC.
¦
CSTEP:  INR     A       ¦set ACC to 1
CRUN:   STA     RUNM    ¦store mode value
        JZ      SRO     ¦if RUNM,bypass enabling RST5.5
        DI              ¦hold intrpts until end routine
        RIM             ¦fetch current mask
        ANI     /06     ¦modify mask
        ORI     /08
        SIM             ¦set new mask, RST5.5 enabled
SRO:    MOV     A,D     ¦check for pending addr param
        ORA     A
        JZ      SR1     ¦none, go from current SAVPC
        SHLD    SAVPC   ¦if pending, store it
SR1:    CALL    CL07    ¦clear display for user
USERPC: POP     PSW     ¦restore user's REGs from stack
        POP     D
```

```
        POP     B
        LHLD    SAVPC   ;load user's PC
        XTHL            ;HL <- orig HL, userPC on stack
        OUT     ARMST5  ;disarm RST5.5 in order to ret
        EI              ;enable interrupts
        RET             ;go to userPC
;
;
;
;
;       COMMAND ADDR PROCESS
;
;       Displays user PC (initial default) and
;         HL-pointed byte.
;       Accepts addr parameter, displaying partial
;         parameter in progress, and HL-pointed byte.
;       Sets up count parameter in reg D:
;         0--addr key not preceeding.
;         FB-FF--addr key preceeding, value is
;                complement of number of hex keys.
;
CADDR:  CALL    DLPC    ;HL = user PC, and display it
        CALL    DLHLP   ;display HL-pointed byte
        CALL    GETW    ;get 4-digit hex value (word)
        CPI     CLEAR   ;terminated with CLEAR key?
        JZ      CADDR   ;yes, repeat display and fetch
        MOV     A,D     ;get key count
        ORA     A
        JNZ     AD1     ;non-zero count, bypass next
        LHLD    SAVPC   ;zero count, use default
AD1:    CMA             ;complement key count
        MOV     D,A     ;save it in reg D
        MOV     A,B     ;propagate terminator key value
        JMP     GC2     ;continue at comm process entry
;
;
;
;
;       COMMAND REG PROCESS
;
;       Displays user PC in leftmost leds.
;       Gets a key and displays corresponding
;         register with value.
;       Accepts a byte from hex keys and alters
;         corresponding register value.
;       Also accepts NEXT key and advances to
;         next register in alphabetic order.
;
CREG:   CALL    DLPC    ;HL = user PC, and display it
        CALL    KEY     ;get a hex keypad closure
        JNC     GC1     ;if command key, go comm entry
        CPI     /08     ;valid register-name?
        JNC     SETRG   ;yes, bypass ERROR call
        MVI     E,/05   ;else, load error code
        CALL    ERROR   ;and call ERROR
        JMP     GETCOM  ;return for next command
```

```
SETRG:  STA     DSPLM   ¦set register display mode
CRG1:   CALL    DLREG   ¦display register and value
        PUSH    H       ¦save ram addr of reg on stack
        CALL    GETB    ¦get a byte from hex keys
        MOV     C,L     ¦save entered value in C
        POP     H       ¦restore ram addr of reg
        CPI     CLEAR   ¦was delimiter a CLEAR key?
        JZ      CRG1    ¦yes, repeat display and fetch
        MOV     A,D     ¦retrieve key count
        ORA     A
        JZ      CRG2    ¦none, bypass update
        MOV     M,C     ¦any keys, update reg in ram
CRG2:   MOV     A,B     ¦propagate terminator key value
        CPI     NEXT    ¦was deliminator a NEXT key?
        JNZ     GC1     ¦no, go command process entr
        LDA     DSPLM   ¦yes, get register key value
        ADI     /F9     ¦point to next reg, wrap-around
        ORI     /08
        JMP     SETRG   ¦continue display and fetch
¦
¦
¦
¦       COMMAND MEM PROCESS
¦
¦       Displays an address and corresponding ram value.
¦
¦       If preceding address parameter contained only one
¦         hex key, that value is taken as a register pair:
¦           1--stack pointer
¦           2--top-of-stack, i.e. the SP-pointed word
¦           8--HL
¦           B--BC
¦           D--DE
¦
¦       If preceding address parameter did not contain
¦         exactly one hex key, that address value is used.
¦
¦       Additionally displays the register-pair name if that
¦         mode was selected to construct the address.
¦
¦       Accepts byte parameter to alter pointed ram.
¦       If address points to rom or non-existent memory,
¦         exits to ERROR display routine.
¦
CMEM:   STAX    B       ¦set DISPLM to HL pointed format
        ADD     D       ¦get key-count and set flags
        JZ      GETPTR  ¦none, use existing RAM pointer
        CPI     /FE     ¦exactly one entered?
        JNZ     SETPTR  ¦no, use ADDR param as entered
        MOV     A,L     ¦yes, use key-value as reg-pair
        LXI     H,RGPTBL¦point to register-pair table
        MVI     C,/05   ¦init table entry counter
CMM1:   CMP     M       ¦compr key to 1st byte of entry
        INX     H       ¦point to display image bytes
```

```
         JZ      CMM2      ¦key match, go reg-pair display
         INX     H         ¦else, point to next entry
         INX     H
         DCR     C         ¦count down loop control
         JNZ     CMM1      ¦if not zero, check next entry
                           ¦if zero and no key match,
         MVI     E,/03     ¦load error code in register E
         CALL    ERROR     ¦go display error
         JMP     GETCOM    ¦loop back for new process
CMM2:    MOV     D,M       ¦get 1st displ-image byte in D
         INX     H         ¦point to 2nd displ-image byte
         MOV     L,M       ¦get it in L
         MVI     H,/05     ¦mov 5 to H
         CALL    DSPLAY    ¦display 2nd image byte in LED5
         MOV     L,D       ¦copy first image byte in L
         MVI     H,/04     ¦mov 4 to H
         CALL    DSPLAY    ¦display 1st image byte in LED4
         LHLD    SAVSP     ¦HL= saved stack pointer
         MVI     D,0       ¦prep MSB of D pair for DAD op
         MOV     A,C       ¦get the residual loop counter
         CPI     /05       ¦was it 5, i.e. 1/P key?
         JZ      CMM3      ¦yes, bypass.
         ADD     C         ¦no, compute 2*loop-count
         MOV     E,A       ¦put it in LSB of DE pair
         DAD     D         ¦offset from saved stack ptr.
         MOV     E,M       ¦get LSB of HL-pointed reg pair
         INX     H         ¦move pointer to MSB
         MOV     D,M       ¦get MSB of HL-pointed reg pair
         XCHG              ¦prep result in HL for RAM ptr.
         JMP     SETPTR    ¦continue with calc. addr param
CMM3:    MVI     E,/08     ¦user SP is 8 more than SAVSP
         DAD     D         ¦make the diff. transparent
SETPTR:  SHLD    RAMPTR    ¦store addr param as RAM ptr.
GETPTR:  LHLD    RAMPTR    ¦retrieve RAM pointer
         CALL    DLHL      ¦display RAM ptr. in left LEDs
         CALL    DLREG     ¦displ HL-pointed in right LEDs
         CALL    GETB      ¦get a 2-digit hex value(byte)
         CPI     CLEAR     ¦terminated with clear key?
         JZ      GETPTR    ¦yes, repeat display and fetch
         DCR     D         ¦set minus flag if key count=0
         MOV     A,L       ¦get entered value
         LHLD    RAMPTR    ¦retrieve RAM pointer
         JM      CMM4      ¦if no keys, bypass update
         MOV     M,A       ¦else, put entered value in RAM
         CMP     M         ¦was it stored as entered?
         JZ      CMM4      ¦yes, continue
         MVI     E,/04     ¦else, load error code
         CALL    ERROR     ¦and display error
         JMP     GETCOM    ¦loop back for new process
CMM4:    MOV     A,B       ¦propagate terminator key value
         JMP     GC1       ¦go command process entry
¦
¦
¦
```

```
;
;       COMMAND ERROR-RESET PROCESS
;
;       Resets error state.
;
CERST:  CALL    ERESET  ;call sub-routine to do the job
        JMP     GETCOM  ;and return
;
;
;
;
;       COMMAND REAL-TIME PROCESSOR CLEAR PROCESS
;
;               Clears RTP by calling RPCLR sub-routine.
;
CRPCLR: CALL    RPCLR
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND VECTOR SET PROCESS
;
;       Loads vector in RTP, if valid conditions exist,
;         from RPVECT in CBLOCK thru VECTOR sub-routine.
;       Invalid conditions are;
;
;       1) RPVECT and RPBRKD both contain zeros.
;       2) RPVECT and RPBRKD contain values such that the
;               computed new vector overflows the boundary.
;
;         Under these conditions, RPVECT is left unchanged
;          (computed new vector not set) and error code '09'
;          is flagged.
;       If valid conditions exist, a new vector is computed,
;         set in RPVECT, and displayed in following manner;
;         1) If RPBRKD contained a zero, new vector is one
;               less than that currently loaded.
;         2) If RPBRKD contained non-zero depth, new vector
;               is offset from the current vector by the
;               specified depth.
;
CVECT:  PUSH    H       ;save pending addr parameter
        PUSH    D       ;if any
        CALL    CL07    ;clear LEDs for new display
        LDA     RPVECT  ;fetch current vector
        MOV     E,A     ;copy
        MVI     D,/01   ;point to LED#1
        CALL    DLADE   ;display current vector(LED0-1)
        LDA     RPBRKD  ;fetch breakpoint depth
        ORA     A       ;check if zero?
        JNZ     CV1     ;no,compute new vector
        DCR     E       ;else, decrement vector
        JM      CV2     ;if bottom end, go error
        MOV     A,E     ;else, proceed further
        JMP     CV3
CV1:    ADD     E       ;new vector = offset thru depth
```

```
        JNC     CV3     ;if no overflow, proceed
CV2:    MVI     E,/09   ;load error code
        CALL    ERROR   ;display error
        JMP     CV4     ;amd exit
CV3:    CALL    VECTOR  ;set current vector
        STA     RPVECT  ;update vector w/computed value
        CALL    DLA     ;display new vector on LED6-7
CV4:    POP     D       ;if any,
        POP     H       ;restore address parameter
        JMP     GETCOM  ;return for next command
;
;
;
;       COMMAND REAL-TIME PROCESSOR RUN PROCESS
;
;       If no depth, RTP is set in RUN mode thru RPRUN subr.
;       If RPBRKD contains non-zero depth, RTP is single
;         stepped thru the depth.
;
CRPRUN: LDA     RPBRKD  ;fetch breakpoint depth
        ORA     A       ;check if zero
        CZ      RPRUN   ;yes, set RUN mode thru RPRUN
        JZ      CRPN2   ;and exit
CRPN1:  CALL    RPSING  ;no, then single step
        DCR     A       ;check if done
        JNZ     CRPN1   ;no, then loop till done
CRPN2:  JMP     GETCOM  ;return for next command
;
;
;
;       COMMAND REAL-TIME PROCESSOR BREAKPOINT RUN PROCESS
;
;       If no address parameter preceded, displays existing
;         vector and breakpoint depth.
;       If address parameter preceded, displays "BP." and
;         awaits depth parameter.
;       If clear entered for depth parameter, default vector
;         (from RPSTRT in CBLOCK) and zero depth are set
;         and displayed.
;       If valid terminator on depth parameter, vector and
;         depth are set and displayed.
;
CRPBRK: PUSH    H       ;save pending addr parameter
        CALL    CL07    ;clear LEDs for new display
        MVI     H,/03   ;point to LED#3
        MVI     L,LEDB  ;load lo-case b for display
        CALL    DSPLAY  ;display on LEDs
        INR     H       ;point tonext LED
        MVI     L,LEDP  ;load capital P. for display
        CALL    DSPLAY  ;display on LEDs
        POP     H       ;restore address parameter
        MOV     A,D     ;check for pending addr param
        ORA     A       ;any pending?
        JZ      DLRBK   ;none, display present breakpt.
```

```
         PUSH    H        ;yes, save it again on stack
         MVI     D,/01    ;point to LED#1
         MOV     A,L      ;copy breakpt. addr for display
         CALL    DLADE    ;display breakpt. addr(LED0-1)
         CALL    GETB     ;get a byte from hex keys
         CPI     CLEAR    ;was delimiter a CLEAR key?
         MOV     A,L      ;get entered value in ACC
         POP     H        ;restore address parameter
         JNZ     SETRBP   ;not CLEAR key, set breakpoint
         LDA     RPSTRT   ;yes, fetch default start addr
         MOV     L,A      ;prep to set default vector
         XRA     A        ;prepare to set zero depth
SETRBP:  STA     RPBRKD   ;set breakpoint depth
         MOV     A,L      ;copy vector to be set
         STA     RPVECT   ;set vector
         MOV     A,B      ;check terminator key again
         CPI     CLEAR    ;was it clear key?
         JNZ     GC1      ;no, proc. the terminator key
DLRBK:   LDA     RPVECT   ;fetch vector (breakpt. addr)
         MVI     D,/01    ;point to LED#1
         CALL    DLADE    ;display it on LED#0 and 1
         LDA     RPBRKD   ;fetch the breakpoint depth
         CALL    DLA      ;display it on LED#6 and 7
         JMP     GETCOM   ;return for next command
;
;
;
;
;        COMMAND REAL-TIME PROCESSOR SINGLE STEP PROCESS
;
;        Single-steps RTP by calling RPSING sub-routine.
;
CRPSIN:  PUSH    H        ;save HL
         PUSH    D        ;save DE
         PUSH    B        ;save BC
         PUSH    PSW      ;save AF
         CALL    RPSING   ;single-step RTP
         CALL    CL07     ;clear LEDs for new display
         LDA     MCROPC   ;fetch RTP program counter
         MVI     D,/01    ;point to LED#1
         CALL    DLADE    ;display RTP PC on LED0-1
         IN      DA2RTP   ;read RTP bus-A
         MVI     D,/04    ;point to LED4
         CALL    DLADE    ;display RTP bus-A on LED3-4
         IN      DB2RTP   ;read RTP bus-B
         CALL    DLA      ;display RTP bus-B on LED6-7
         POP     PSW      ;restore AF
         POP     B        ;restore BC
         POP     D        ;restore DE
         POP     H        ;restore HL
         JMP     GETCOM   ;return for next command
;
;
;
;
;        COMMAND REAL-TIME PROCESSOR HOLD PROCESS
```

```
;
;       Turns off the clock to RTP by calling RPHOLD.
;
CRPHLD: CALL    RPHOLD
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-0 PROCESS
;
;       Performs diagnosis-0 by calling DIaGnose0 subroutine.
;
CDIG0:  CALL    DIG0
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-1 PROCESS
;
;       Performs diagnosis-1 by calling DIaGnose1 subroutine.
;
CDIG1:  CALL    DIG1
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-2 PROCESS
;
;       Performs diagnosis-2 by calling DIaGnose2 subroutine.
;
CDIG2:  CALL    DIG2
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-3 PROCESS
;
;       Performs diagnosis-3 by calling DIaGnose3 subroutine.
;
CDIG3:  CALL    DIG3
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-4 PROCESS
;
;       Performs diagnosis-4 by calling DIaGnose4 subroutine.
;
CDIG4:  CALL    DIG4
        JMP     GETCOM  ;return for next command
;
;
;
;
;       COMMAND RTP DIAGNOSIS-5 PROCESS
```

```
;
;       Performs diagnosis-5 by calling DIaGnose5 subroutine.
;
CDIG5:  CALL    DIG5
        JMP     GETCOM  ;return for next command
;
;
;
;       COMMAND RTP DIAGNOSIS-6 PROCESS
;
;       Performs diagnosis-6 by calling DIaGnose6 subroutine.
;
CDIG6:  CALL    DIG6
        JMP     GETCOM  ;return for next command
;
;
;
;       COMMAND RTP DIAGNOSIS-7 PROCESS
;
;       Performs diagnosis-7 by calling DIaGnose7 subroutine.
;
CDIG7:  CALL    DIG7
        JMP     GETCOM  ;return for next command
;
;
;
;       COMMAND RAM TeST PROCESS
;
;       This process is for carrying out RAM memory
;         diagnostics. If the memory is good, message
;         "Good" is displayed on LEDs. Else, defective
;         memory location is displayed alongwith its
;         contents as well as what the contents should
;         actually have been. Display format is as
;         follows;
;
;  LED  0    1    2    3       4    5       6    7
;       defective location    correct value  actual value
;
;       To continue from the breakpoint MEM key should be hit.
;       If any other key is hit, diagnostic process is
;       reinitiated from the very start, for the same
;       arguments.
;
;       Expects start-address in DE and end-address in BC.
;       These registers are returned unchanged.
;
;       If any attempt is made to check PROM area (loc /0000
;         thru /0FFF ) code 7 error occurs.
;
; Constraints: If low 32 bytes in page1 ( first RAM block of 4K
;       - loc. /1000 thru /1FFF ) is included in the check
;       region, top 32 bytes of the page should be excluded.
;
```

```
;        RAM MAP :  /1000 - /1FFF, /8000 - /87FF, /8800 - /8FFF
;
CMTEST: DI                  ;disable intr during this proc.
        LXI     H,SAVDE     ;fetch test parameters
        MOV     E,M         ;reg pair DE to hold start addr
        INX     H           ;point to next location
        MOV     D,M
        INX     H           ;point to next location
        MOV     C,M         ;reg pair BC to hold end addr
        INX     H           ;point to next location
        MOV     B,M
CMT1:   MOV     H,D         ;init HL with start address
        MOV     L,E
        MOV     A,D         ;establish region for stack loc
        CPI     /20         ;check area on page1?
        JNC     POKE3       ;no, use std. stack location
        CPI     /10         ;yes, test space include PROMs?
        JNC     POKE1       ;no, proceed further
        SHLD    PANBOX      ;yes, save HL in PANBOX
        MVI     E,/07       ;and go error
        CALL    ERROR
        JMP     GETCOM      ;return for next command
POKE1:  JNZ     POKE2       ;if test space not in low area
        MOV     A,E         ;region in low area of page1
        CPI     /20         ;region includes low 32 bytes?
        JNC     POKE2       ;no, then go to POKE2
        LXI     SP,/1FFF    ;yes, loc stack on top of page
        JMP     POKE3       ;and proceed further
POKE2:  SPHL                ;locate stack below test space
POKE3:  XRA     A           ;clear ACC
        MOV     M,A         ;store 0 in HL-pointed location
        CMP     M           ;check if memory loc. cleared
        CNZ     MEMERR      ;no, then display error
        INX     H           ;point to next location
        MOV     A,L         ;check if end of region
        CMP     C
        JNZ     POKE3       ;no, then loop and continue
        MOV     A,H
        CMP     B
        JNZ     POKE3       ;no, then loop and continue
        MOV     H,D         ;yes, then begin a new pass
        MOV     L,E
POKE4:  XRA     A           ;to check for addressing error
        CMP     M           ;check addressing error
        CNZ     MEMERR      ;yes, display error
        INR     A           ;no, start next procedure
POKE5:  MOV     M,A         ;store new pattern
        CMP     M           ;did it store correctly?
        CNZ     MEMERR      ;no, display error
        RLC                 ;yes, go for next bit
        JNC     POKE5       ;if not done, loop and continue
        MVI     A,/FF       ;load all ones pattern
        MOV     M,A         ;fill location with all ones
        CMP     M           ;was it stored correctly?
```

```
        CNZ     MEMERR  ¦no, signal error
        INX     H       ¦done, point to next location
        MOV     A,L     ¦check if end of region
        CMP     C
        JNZ     POKE4   ¦no,loop for checking next byte
        MOV     A,H
        CMP     B
        JNZ     POKE4   ¦no,loop for checking next byte
        MOV     H,D     ¦yes, begin a new pass
        MOV     L,E
POKE6:  MVI     A,/FF   ¦now start check for all ones
        CMP     M
        CNZ     MEMERR  ¦no, display error
        INX     H       ¦yes, point to next location
        MOV     A,L     ¦check if end of region
        CMP     C
        JNZ     POKE6   ¦not yet, loop and continue
        MOV     A,H
        CMP     B
        JNZ     POKE6   ¦not yet, loop and continue
        CALL    MSGOOD  ¦check completed and successful
        JMP     CMT1    ¦do it all over again
¦
¦
¦       RPCLR subroutine
¦
¦       Clears RTP by pulling SYSCLR high and then low.
¦       Does not change registers or status.
¦
RPCLR:  PUSH    PSW     ¦save caller's AF
        MVI     A,SCLRH ¦load SYSCLR-hi template
        OUT     C1RTP   ¦pull SYSCLR high
        MVI     A,SCLRL ¦load SYSCLR-lo template
        OUT     C1RTP   ¦pull SYSCLR low
        POP     PSW     ¦restore caller's AF
        RET
¦
¦
¦
¦       VECTOR sub-routine
¦
¦       Expects vector in RPVECT of CBLOCK.
¦       Loads vector in real-time PC thru RTP-INTerrupt.
¦       Does not alter any registers.
¦
VECTOR: PUSH    PSW     ¦save caller's AF
        LDA     RPVECT  ¦fetch vector from CBLOCK
        OUT     DA1RTP  ¦write vector to RTP interface
        MVI     A,RINTH ¦load template for RTP intr.
        OUT     C1RTP   ¦interrupt RTP
        MVI     A,RINTL ¦load template for normal op
        OUT     C1RTP   ¦normalize RTP interrupt
        CALL    RPSING  ¦single-step RTP
        CALL    RPSING  ¦single-step RTP
```

```
        CALL    RPSING  ;single-step RTP
        POP     PSW     ;restore caller's AF
        RET
;
;
;
;       RPRUN   sub-routine
;
;       Sets RTP in RUN mode by pulling CLKMOD
;          low and RTPRUN high.
;       Also set RTP Data Interface for RUN mode.
;       Does not change registers or status.
;
RPRUN:  PUSH    PSW     ;save caller's AF
        MVI     A,SCLRL ;load SYSCLR-lo template
        OUT     C1RTP   ;pull SYSCLR line low
        MVI     A,DSTEPL;load template
        OUT     C2RTP   ;set DATASTEP' line high
        MVI     A,DFLOW ;load template
        OUT     C2RTP   ;set DATAHOLD' line high
        MVI     A,IGNDL ;load template
        OUT     C2RTP   ;set IGNORDTA' line high
        MVI     A,CLKMDL;load template
        OUT     C1RTP   ;set CLKMOD line low
        MVI     A,RRUNH ;load template
        OUT     C1RTP   ;set RTPRUN line high
        POP     PSW     ;restore caller's AF
        RET
;
;
;
;       RPSING sub-routine
;
;       Single-steps RTP by twiddling CLKMOD
;       and RTPRUN lines.
;       Does not change any registers or status.
;
RPSING: PUSH    PSW     ;save caller's AF
        MVI     A,SCLRL ;load SYSCLR-lo template
        OUT     C1RTP   ;pull SYSCLR line low
        MVI     A,RRUNL ;load template
        OUT     C1RTP   ;set RTPRUN line low
        MVI     A,CLKMDH;load template
        OUT     C1RTP   ;set CLKMOD line high
        MVI     A,RRUNH ;load template
        OUT     C1RTP   ;set RTPRUN line high
        MVI     A,RRUNL ;reset RTPRUN
        OUT     C1RTP
        MVI     A,CLKMDL;reset CLKMOD
        OUT     C1RTP
        POP     PSW     ;restore caller's AF
        RET
;
;
```

```
;
;       RPHOLD sub-routine
;
;       Cuts off clock to RTP.
;       Does not change registers or status.
;
RPHOLD: PUSH    PSW     ;save caller's AF
        MVI     A,RRUNL ;load template
        OUT     C1RTP   ;set RTPRUN line low
        POP     PSW     ;restore caller's AF
        RET
;
;
;
;       MEMERR sub-routine
;
;       Displays defective memeory location in
;         LED0 thru LED3, contents of ACC in
;         LED4 & LED5, contents of MEMory in
;         LED6 & LED7 , and then waits for a key
;         to be depressed. If the depressed key is
;         MEM, then memory diagnostics is resumed
;         from the broken point. Any other key
;         reinitiates diagnostics from the start
;         for the same arguments.
;       Expects HL pointing to the defective location
;         and ACC contents to be the same as was used
;         for memory check.
;       Returns BC, DE and HL unchanged.
;
MEMERR: PUSH    H       ;save caller's HL
        PUSH    D       ;save caller's DE
        PUSH    B       ;save caller's BC
        MVI     D,/05   ;point to LED5
        CALL    DLADE   ;display ACC in LED4 and LED5
        CALL    DLHL    ;display HL in LED0 thru LED3
        CALL    DLHLP   ;display HL-pointed in LED6-7
        CALL    KEY     ;wait for key to be depressed
        CPI     /19     ;key value matches RAMTST ?
        POP     B       ;restore caller's BC
        POP     D       ;restore caller's DE
        POP     H       ;restore caller's HL
        RZ              ;key matched,cont. from brkpt
        JMP     CMT1    ;else, start testing again
;
;
;
;       MeSsage-GOOD sub-routine
;
;       Displays "Good" in LED0 thru LED3
;
MSGOOD: PUSH    H       ;save caller's HL
        PUSH    PSW     ;save caller's AF
        MVI     H,0     ;point to LED0
```

```
        MVI     L,LEDG  ;load pattern for capital G
        CALL    DSPLAY  ;display 'G'
        INR     H       ;point to next LED
        MVI     L,LEDO  ;load pattern for locase o
        CALL    DSPLAY  ;display 'o'
        INR     H       ;point to next LED
        CALL    DSPLAY  ;display another 'o'
        INR     H       ;point to next LED
        MVI     L,LEDD  ;load pattern for locase d
        CALL    DSPLAY  ;display 'd'
        POP     PSW     ;restore caller's AF
        POP     H       ;restore caller's HL
        RET
;
;
;
;       MONITOR COMMAND TABLE
;
MONTBL: .WORD   MGIVER  ;command = /00, qualifier = /00
        .WORD   MGET    ;command = /01, qualifier = /xx
        .WORD   MPUT    ;command = /02, qualifier = /xx
        .WORD   MEDMP   ;command = /03, qualifier = /00
        .WORD   MERST   ;command = /04, qualifier = /00
        .WORD   MRUN    ;command = /05, qualifier = /00
        .WORD   MHOLD   ;command = /06, qualifier = /00
        .WORD   MSDMP   ;command = /07, qualifier = /00
        .WORD   MINIT   ;command = /08, qualifier = /00
        .WORD   MDIG0   ;command = /09, qualifier = /00
        .WORD   MDIG1   ;command = /0A, qualifier = /00
        .WORD   MDIG2   ;command = /0B, qualifier = /00
        .WORD   MDIG3   ;command = /0C, qualifier = /00
        .WORD   MDIG4   ;command = /0D, qualifier = /00
        .WORD   MDIG5   ;command = /0E, qualifier = /00
        .WORD   MDIG6   ;command = /0F, qualifier = /00
        .WORD   MDIG7   ;command = /10, qualifier = /00
;
;
;
;       ReSTart 6.5 INTERRUPT HANDLER ROUTINE
;
;       Receives command code thru Control Port on HOP
;       Interface, which is used as a vector into
;       MONitor-TaBLe to provide address of the corresponding
;       service routine.
;
R6HAND: PUSH    H       ;save HL
        PUSH    D       ;save DE
        PUSH    B       ;save BC
        PUSH    PSW     ;save AF
        LXI     D,0     ;clear DE for DAD operation
        LXI     H,MONTBL;load monitor table origin
        IN      DB1HOS  ;control port - low byte
        ADD     A       ;2X comm code for word offset
        JNC     R6H1    ;skip next if not carry
```

```
        INR     D
R6H1:   MOV     E,A      |DE=word offset in mon-table
        DAD     D        |HL=pointer to service routine
        MOV     E,M      |fetch low byte of address
        INX     H        |point to next location
        MOV     H,M      |fetch high byte of address
        MOV     L,E      |HL=address of service routine
        PCHL             |branch to service routine
|
|
|
|       ReSTart 6.5 EXIT PROCEDURE
|
|       This procedure is common to all service routines.
|
R6EXIT: POP     PSW
        POP     B
        POP     D
        POP     H
        EI
        RET
|
|
|
|       SERVICE ROUTINE No. 0
|
|       Outputs Version and Sub-Version numbers.
|
MGIVER: IN      DB2HOS   |dummy read of qualifier port
        MVI     A,SUBVER |load subversion number
        OUT     DA1HOS   |output to HOP as low byte
        MVI     A,VER    |load version number
        OUT     DA2HOS   |output to HOP as high byte
        JMP     R6EXIT   |exit
|
|
|
|       SERVICE ROUTINE No. 1
|
|       Fetches a record of specified length from specified
|       ICMC address and outputs to HOP word by word.
|
MGET:   IN      DA1HOS   |get source address low byte
        MOV     L,A      |init reg L with it
        IN      DA2HOS   |get source address high byte
        MOV     H,A      |init reg H with it
        CPI     /7F      |check if RTP mem referenced
        CP      RPHOLD   |yes, set RTP in HOLD mode
        IN      DB2HOS   |get rec length from qual port
        MOV     B,A      |save in reg B
        INR     B
MGET1:  DCR     B        |check if done
        JZ      R6EXIT   |yes, exit
        MOV     A,M      |no, get current low byte first
```

```
        OUT     WAITOT  ;wait if last data not picked
        OUT     DA1HOS  ;output low data byte to HOP
        INX     H       ;point to next location
        MOV     A,M     ;get current high byte
        OUT     DA2HOS  ;output high data byte to HOP
        INX     H       ;point to next location
        JMP     MGET1   ;loop till done
;
;
;
;       SERVICE ROUTINE No. 2
;
;       Loads a record of specified length at specified
;       destination ICMC address from HOP word by word.
;
MPUT:   MVI     E,/04   ;init reg E in case of error
        IN      DA1HOS  ;get destination addr low byte
        MOV     L,A     ;init reg L with it
        IN      DA2HOS  ;get destination addr high byte
        MOV     H,A     ;init reg H with it
        CPI     /7F     ;check if RTP memory referenced
        CP      RPHOLD  ;yes, set RTP in HOLD mode
        IN      DB2HOS  ;get rec length from qual port
        MOV     B,A     ;save it in reg B
        INR     B
MPUT1:  DCR     B       ;check if done
        JZ      R6EXIT  ;yes, exit
        IN      WAITIN  ;no, wait if data not available
        IN      DA1HOS  ;get data low byte from HOP
        MOV     M,A     ;store in current location
        CMP     M       ;check if correctly written
        CNZ     ERROR   ;no, signal error
        INX     H       ;point to next location
        IN      DA2HOS  ;get data high byte from HOP
        MOV     M,A     ;store in current location
        CMP     M       ;check if correctly written
        CNZ     ERROR   ;no, signal error
        INX     H       ;point to next location
        JMP     MPUT1   ;loop till done
;
;
;
;       SERVICE ROUTINE No. 3
;
;       Transmits error code to HOP.
;
MEDMP:  IN      DB2HOS  ;dummy read of qualifier port
        LDA     ERCODE  ;load error code
        OUT     DA1HOS  ;output as low byte to HOP
        XRA     A       ;load a zero
        OUT     DA2HOS  ;output as high byte to HOP
        JMP     R6EXIT  ;exit
;
;
```

```
;
;       SERVICE ROUTINE No. 4
;
;       Resets error condition by calling ERESET.
;
MERST:  IN      DB2HOS  ;dummy read of qualifier port
        CALL    ERESET  ;let others do the job
        JMP     R6EXIT  ;exit
;
;
;
;       SERVICE ROUTINE No. 5
;
;       Sets RTP in RUN mode by calling RTPRUN.
;
MRUN:   IN      DB2HOS  ;dummy read of qualifier port
        CALL    RPRUN   ;let others do the job
        JMP     R6EXIT  ;exit
;
;
;
;       SERVICE ROUTINE No. 6
;
;       Sets RTP in HOLD mode by calling RPHOLD.
;
MHOLD:  IN      DB2HOS  ;dummy read of qualifier port
        CALL    RPHOLD  ;let others do the job
        JMP     R6EXIT  ;exit
;
;
;
;       SERVICE ROUTINE No. 9
;
;       Performs diagnostic procedure no. 0 by calling DIGO.
;
MDIGO:  IN      DB2HOS  ;dummy read of qualifier port
        CALL    DIGO    ;let others do the job
        JMP     R6EXIT  ;exit
;
;
;
;       SERVICE ROUTINE No. 10
;
;       Performs diagnostic procedure no. 1 by calling DIG1.
;
MDIG1:  IN      DB2HOS  ;dummy read of qualifier port
        CALL    DIG1    ;let others do the job
        JMP     R6EXIT  ;exit
;
;
;
;       SERVICE ROUTINE No. 11
;
;       Performs diagnostic procedure no. 2 by calling DIG2.
```

```
!
MDIG2:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG2    |let others do the job
        JMP     R6EXIT  |exit
!
!
!
!
!       SERVICE ROUTINE No. 12
!
!       Performs diagnostic procedure no. 3 by calling DIG3
!
MDIG3:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG3    |let others do the job
        JMP     R6EXIT  |exit
!
!
!
!       SERVICE ROUTINE No. 13
!
!       Performs diagnostic procedure no. 4 by calling DIG4.
!
MDIG4:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG4    |let others do the job
        JMP     R6EXIT  |exit
!
!
!
!       SERVICE ROUTINE No. 14
!
!       Performs diagnostic procedure no. 5 by calling DIG5.
!
MDIG5:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG5    |let others do the job
        JMP     R6EXIT  |exit
!
!
!
!       SERVICE ROUTINE No. 15
!
!       Performs diagnostic procedure no. 6 by calling DIG6.
!
MDIG6:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG6    |let others do the job
        JMP     R6EXIT  |exit
!
!
!
!       SERVICE ROUTINE No. 16
!
!       Performs diagnostic procedure no. 7 by calling DIG7.
!
MDIG7:  IN      DB2HOS  |dummy read of qualifier port
        CALL    DIG7    |let others do the job
        JMP     R6EXIT  |exit
```

```
;
;
;       SERVICE ROUTINE No. 7
;
;       Transmits RTP status to HOP.
;
MSDMP:  IN      DB2HOS  ;dummy read of qualifier port
        LDA     STATUS  ;load status from CBLOCK
        OUT     DA1HOS  ;output as low byte to HOP
        XRA     A       ;load a zero
        OUT     DA2HOS  ;output as high byte to HOP
        JMP     R6EXIT  ;exit
;
;
;       SERVICE ROUTINE No. 8
;
;       Performs initialization of RTP by calling
;       appropriate routines.
;
MINIT:  IN      DB2HOS  ;dummy read of qualifier port
        MVI     B,/10   ;load reg B with 16
        XRA     A       ;clear ACC
        LXI     H,/8000 ;point to base of MICPGM memory
MINIT1: MOV     M,A     ;clear memory
        INX     H       ;point to next location
        DCR     B       ;loop 16 times
        JNZ     MINIT1
        LXI     H,/8003 ;point 1st microprog cont. inst.
        MVI     M,/16   ;init with JVEC instruction
        MVI     A,/02   ;load MICPGM start address
        STA     RPVECT  ;save it as vector
        CALL    VECTOR  ;vector RTP to start address
        CALL    RPRUN   ;set RTP in RUN mode
        LDA     ERCODE  ;fetch error code
        CPI     /01     ;check if due to lack of init
        CZ      ERESET  ;yes, then clear error state
        JMP     R6EXIT  ;exit
;
;
;       ReSTart 7.5 INTERRUPT HANDLER ROUTINE
;
R7HAND: PUSH    D       ;save DE on stack
        PUSH    PSW     ;save AF on stack
        MVI     A,ICMBSY;load ICMRDY turn-off template
        OUT     C2RTP   ;init ICMRDY to off
        MVI     A,IGNDH ;load template
        OUT     C2RTP   ;set IGNORDTA' line low
        MVI     E,/01   ;set error code 1
        CALL    ERROR   ;signal need for initialization
        IN      DB1RTP  ;fetch new status
        STA     STATUS  ;save new status in CBLOCK
```

```
        POP     PSW       |restore AF from stack
        POP     D         |restore DE from stack
        EI                |enable interrupts
        RET
|
|
|
|       TRAP INTERRUPT HANDLER ROUTINE
|
TRHAND: PUSH    D         |save  DE on stack
        MVI     E,/0A     |load error code
        CALL    ERROR     |display error
        POP     D         |restore DE
        OUT     TRACK     |rearm watchdog
        EI                |enable interrupts
        RET
|
|
        .=/0800           |origin for 3rd PROM chip
|
|  DIAGNOSTIC ROUTINES
|
|  These routines transfer the diagnostic microprograms
|  to RTP's memory, and sets RTP to execute it.
|
DIGO:   PUSH    PSW       |save callers's AF
        PUSH    H         |save caller's HL
        PUSH    D         |save caller's DE
        PUSH    B         |save caller's BC
        LXI     D,MPORG   |load source pointer
        LXI     B,/8000   |load destination pointer
        MVI     L,/10     |load byte count
D01:    LDAX    D         |fetch data from source
        STAX    B         |store in destination
        INX     D         |increment source pointer
        INX     B         |increment destination pointer
        DCR     L         |decrement byte count
        JNZ     D01       |if not done, loop
        LXI     D,MPGM0   |load source pointer
        LXI     B,/8010   |load destination pointer
        LXI     H,/300    |load byte count
D02:    LDAX    D         |fetch data from source
        STAX    B         |store in destination
        INX     D         |increment source pointer
        INX     B         |increment destination pointer
        DCX     H         |decrement byte count
        MOV     A,L       |check if done
        ORA     A
        JNZ     D02       |if not, loop
        MOV     A,H
        ORA     A
        JNZ     D02
        STA     RPBRKD    |if done, clear depth param
        MVI     A,/02     |load microprogram start addr
```

```
        STA     RPVECT  ;set it as param
        CALL    VECTOR  ;vector RTP to this start addr
        CALL    RPRUN   ;set RTP in RUN mode
        POP     B       ;restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
;
;
;
DIG1:   PUSH    PSW     ;save callers's AF
        PUSH    H       ;save caller's HL
        PUSH    D       ;save caller's DE
        PUSH    B       ;save caller's BC
        LXI     D,MPORG ;load source pointer
        LXI     B,/8000 ;load destination pointer
        MVI     L,/10   ;load byte count
D11:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
        INX     B       ;increment destination pointer
        DCR     L       ;decrement byte count
        JNZ     D11     ;if not done, loop
        LXI     D,MPGM1 ;load source pointer
        LXI     B,/8010 ;load destination pointer
        LXI     H,/300  ;load byte count
D12:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
        INX     B       ;increment destination pointer
        DCX     H       ;decrement byte count
        MOV     A,L     ;check if done
        ORA     A
        JNZ     D12     ;if not, loop
        MOV     A,H
        ORA     A
        JNZ     D12
        STA     RPBRKD  ;if done, clear depth param
        MVI     A,/02   ;load microprogram start addr
        STA     RPVECT  ;set it as param
        CALL    VECTOR  ;vector RTP to this start addr
        CALL    RPRUN   ;set RTP in RUN mode
        POP     B       ;restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
;
;
;
DIG2:   PUSH    PSW     ;save callers's AF
        PUSH    H       ;save caller's HL
        PUSH    D       ;save caller's DE
```

```
        PUSH    B       |save caller's BC
        LXI     D,MPORG |load source pointer
        LXI     B,/8000 |load destination pointer
        MVI     L,/10   |load byte count
D21:    LDAX    D       |fetch data from source
        STAX    B       |store in destination
        INX     D       |increment source pointer
        INX     B       |increment destination pointer
        DCR     L       |decrement byte count
        JNZ     D21     |if not done, loop
        LXI     D,MPGM2 |load source pointer
        LXI     B,/8010 |load destination pointer
        LXI     H,/300  |load byte count
D22:    LDAX    D       |fetch data from source
        STAX    B       |store in destination
        INX     D       |increment source pointer
        INX     B       |increment destination pointer
        DCX     H       |decrement byte count
        MOV     A,L     |check if done
        ORA     A
        JNZ     D22     |if not, loop
        MOV     A,H
        ORA     A
        JNZ     D22
        STA     RPBRKD  |if done, clear depth param
        MVI     A,/02   |load microprogram start addr
        STA     RPVECT  |set it as param
        CALL    VECTOR  |vector RTP to this start addr
        CALL    RPRUN   |set RTP in RUN mode
        POP     B       |restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
|
|
|
DIG3:   PUSH    PSW     |save callers's AF
        PUSH    H       |save caller's HL
        PUSH    D       |save caller's DE
        PUSH    B       |save caller's BC
        LXI     D,MPORG |load source pointer
        LXI     B,/8000 |load destination pointer
        MVI     L,/10   |load byte count
D31:    LDAX    D       |fetch data from source
        STAX    B       |store in destination
        INX     D       |increment source pointer
        INX     B       |increment destination pointer
        DCR     L       |decrement byte count
        JNZ     D31     |if not done, loop
        LXI     D,MPGM3 |load source pointer
        LXI     B,/8010 |load destination pointer
        LXI     H,/300  |load byte count
D32:    LDAX    D       |fetch data from source
```

```
        STAX    B       ¦store in destination
        INX     D       ¦increment source pointer
        INX     B       ¦increment destination pointer
        DCX     H       ¦decrement byte count
        MOV     A,L     ¦check if done
        ORA     A
        JNZ     D32     ¦if not, loop
        MOV     A,H
        ORA     A
        JNZ     D32
        STA     RPBRKD  ¦if done, clear depth param
        MVI     A,/C2   ¦load microprogram start addr
        STA     RPVECT  ¦set it as param
        CALL    VECTOR  ¦vector RTP to this start addr
        CALL    RPRUN   ¦set RTP in RUN mode
        POP     B       ¦restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
¦
¦
¦
DIG4:   PUSH    PSW     ¦save callers's AF
        PUSH    H       ¦save caller's HL
        PUSH    D       ¦save caller's DE
        PUSH    B       ¦save caller's BC
        LXI     D,MPORG ¦load source pointer
        LXI     B,/8000 ¦load destination pointer
        MVI     L,/10   ¦load byte count
D41:    LDAX    D       ¦fetch data from source
        STAX    B       ¦store in destination
        INX     D       ¦increment source pointer
        INX     B       ¦increment destination pointer
        DCR     L       ¦decrement byte count
        JNZ     D41     ¦if not done, loop
        LXI     D,MPGM4 ¦load source pointer
        LXI     B,/8010 ¦load destination pointer
        LXI     H,/300  ¦load byte count
D42:    LDAX    D       ¦fetch data from source
        STAX    B       ¦store in destination
        INX     D       ¦increment source pointer
        INX     B       ¦increment destination pointer
        DCX     H       ¦decrement byte count
        MOV     A,L     ¦check if done
        ORA     A
        JNZ     D42     ¦if not, loop
        MOV     A,H
        ORA     A
        JNZ     D42
        STA     RPBRKD  ¦if done, clear depth param
        MVI     A,/02   ¦load microprogram start addr
        STA     RPVECT  ¦set it as param
        CALL    VECTOR  ¦vector RTP to this start addr
```

```
        CALL    RPRUN   ¦set RTP in RUN mode
        POP     B       ¦restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
¦
¦
¦
DIG5:   PUSH    PSW     ¦save callers's AF
        PUSH    H       ¦save caller's HL
        PUSH    D       ¦save caller's DE
        PUSH    B       ¦save caller's BC
        LXI     D,MPORG ¦load source pointer
        LXI     B,/8000 ¦load destination pointer
        MVI     L,/10   ¦load byte count
D51:    LDAX    D       ¦fetch data from source
        STAX    B       ¦store in destination
        INX     D       ¦increment source pointer
        INX     B       ¦increment destination pointer
        DCR     L       ¦decrement byte count
        JNZ     D51     ¦if not done, loop
        LXI     D,MPGM5 ¦load source pointer
        LXI     B,/8010 ¦load destination pointer
        LXI     H,/300  ¦load byte count
D52:    LDAX    D       ¦fetch data from source
        STAX    B       ¦store in destination
        INX     D       ¦increment source pointer
        INX     B       ¦increment destination pointer
        DCX     H       ¦decrement byte count
        MOV     A,L     ¦check if done
        ORA     A
        JNZ     D52     ¦if not, loop
        MOV     A,H
        ORA     A
        JNZ     D52
        STA     RPBRKD  ¦if done, clear depth param
        MVI     A,/02   ¦load microprogram start addr
        STA     RPVECT  ¦set it as param
        CALL    VECTOR  ¦vector RTP to this start addr
        CALL    RPRUN   ¦set RTP in RUN mode
        POP     B       ¦restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
¦
¦
¦
DIG6:   PUSH    PSW     ¦save callers's AF
        PUSH    H       ¦save caller's HL
        PUSH    D       ¦save caller's DE
        PUSH    B       ¦save caller's BC
        LXI     D,MPORG ¦load source pointer
```

```
        LXI     B,/8000 ;load destination pointer
        MVI     L,/10   ;load byte count
D61:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
        INX     B       ;increment destination pointer
        DCR     L       ;decrement byte count
        JNZ     D61     ;if not done, loop
        LXI     D,MPGM6 ;load source pointer
        LXI     B,/8010 ;load destination pointer
        LXI     H,/300  ;load byte count
D62:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
        INX     B       ;increment destination pointer
        DCX     H       ;decrement byte count
        MOV     A,L     ;check if done
        ORA     A
        JNZ     D62     ;if not, loop
        MOV     A,H
        ORA     A
        JNZ     D62
        STA     RPBRKD  ;if done, clear depth param
        MVI     A,/02   ;load microprogram start addr
        STA     RPVECT  ;set it as param
        CALL    VECTOR  ;vector RTP to this start addr
        CALL    RPRUN   ;set RTP in RUN mode
        POP     B       ;restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
;
;
;
;
DIG7:   PUSH    PSW     ;save callers's AF
        PUSH    H       ;save caller's HL
        PUSH    D       ;save caller's DE
        PUSH    B       ;save caller's BC
        LXI     D,MPORG ;load source pointer
        LXI     B,/8000 ;load destination pointer
        MVI     L,/10   ;load byte count
D71:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
        INX     B       ;increment destination pointer
        DCR     L       ;decrement byte count
        JNZ     D71     ;if not done, loop
        LXI     D,MPGM7 ;load source pointer
        LXI     B,/8010 ;load destination pointer
        LXI     H,/300  ;load byte count
D72:    LDAX    D       ;fetch data from source
        STAX    B       ;store in destination
        INX     D       ;increment source pointer
```

```
        INX     B          ;increment destination pointer
        DCX     H          ;decrement byte count
        MOV     A,L        ;check if done
        ORA     A
        JNZ     D72        ;if not, loop
        MOV     A,H
        ORA     A
        JNZ     D72
        STA     RPBRKD     ;if done, clear depth param
        MVI     A,/02      ;load microprogram start addr
        STA     RPVECT     ;set it as param
        CALL    VECTOR     ;vector RTP to this start addr
        CALL    RPRUN      ;set RTP in RUN mode
        POP     B          ;restore caller's registers
        POP     D
        POP     H
        POP     PSW
        RET
;
;
;
MPORG:  .BYTE   ,,,/16,,,,
        .BYTE   ,,,/1E,,,,
;
;
;
MPGM0:  .BYTE   /0A,/90,,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   /1E,/E0,/01,/1E,/C4,,,
        .BYTE   ,/E0,/C1,/13,,/02,,
        .BYTE   ,,,/1E,,,,
;
;
;
MPGM1:  .BYTE   /0A,/01,,/1E,,,,
        .BYTE   /0B,/10,,/1E,/99,,,
        .BYTE   ,,,/A3,,/08,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,/A0,/C1,/13,,/02,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,/B0,/C0,/13,,/02,,
        .BYTE   ,,,/1E,,,,
;
;
;
MPGM2:  .BYTE   /0A,/01,,/1E,,,,
        .BYTE   /0B,/10,,/1E,/44,,,
        .BYTE   /0C,/90,,/1E,/47,/01,,
        .BYTE   ,,,/1E,,,,
```

```
        .BYTE    ,/B0,/C1,/13,,/02,,
        .BYTE    ,,,/1E,,,,
;
;
;
MPGM3:  .BYTE    /0A,/04,,/1E,,,,
        .BYTE    /0B,/05,,/1E,,,,
        .BYTE    /0C,/06,,/1E,,,,
        .BYTE    /04,/50,,/1E,/86,,,
        .BYTE    ,,,/43,,/0D,,
        .BYTE    ,,,/1E,,,,
        .BYTE    /04,/60,,/1E,/86,,,
        .BYTE    ,,,/43,,/12,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,/40,/C0,/13,,/02,,
        .BYTE    ,,,/1E,,,,
        .BYTE    /05,/60,,/1E,/86,,,
        .BYTE    ,,,/43,,/12,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,/50,/C0,/13,,/02,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,/60,/C0,/13,,/02,,
        .BYTE    ,,,/1E,,,,
;
;
;
MPGM4:  .BYTE    /0A,/C0,/10,/1E,,,,
        .BYTE    /0A,/C0,/10,/1E,,,,
        .BYTE    /0B,/C0,/20,/1E,,,,
        .BYTE    /0B,/C0,/20,/1E,,,,
        .BYTE    /0C,/C0,/30,/1E,,,,
        .BYTE    /0C,/C0,/30,/1E,,,,
        .BYTE    /03,/90,,/1E,/44,/0E,,
        .BYTE    /02,/90,,/1E,/45,/49,,
        .BYTE    /01,/90,,/1E,/47,/EF,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,,,/83,,/13,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,,,/03,,/15,,
        .BYTE    ,,,/1E,,,,
        .BYTE    /1B,/C0,/01,/1E,/C4,,,
        .BYTE    ,/E0,/C1,/13,,/02,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,/A0,/C0,/13,,/02,,
        .BYTE    ,,,/1E,,,,
        .BYTE    ,/B0,/C0,/13,,/02,,
        .BYTE    ,,,/1E,,,,
;
;
;
MPGM5:  .BYTE    /0A,/C0,/10,/1E,,,,
        .BYTE    /0A,/C0,/10,/1E,,,,
        .BYTE    /0B,/C0,/20,/1E,,,,
        .BYTE    /0B,/C0,/20,/1E,,,,
```

```
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /03,/90,,/1E,/44,/23,,
        .BYTE   /01,/90,,/1E,/45,/94,,
        .BYTE   /02,/90,,/1E,/47,/FF,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,,,/83,,/13,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,,,/03,,/15,,
        .BYTE   ,,,/1E,,,,
        .BYTE   /1B,/C0,/01,/1E,/C4,,,
        .BYTE   ,/E0,/C1,/13,,/02,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,/A0,/C0,/13,,/02,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,/B0,/C0,/13,,/02,,
        .BYTE   ,,,/1E,,,,
!
!
!
!
MPGM6:  .BYTE   /0A,/C0,/10,/1E,,,,
        .BYTE   /0A,/C0,/10,/1E,,,,
        .BYTE   /0B,/C0,/20,/1E,,,,
        .BYTE   /0B,/C0,/20,/1E,,,,
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /01,/90,,/1E,/44,/18,,
        .BYTE   /02,/90,,/1E,/45,/2F,,
        .BYTE   /03,/90,,/1E,/45,/51,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,,,/03,,/11,,
        .BYTE   ,,,/1E,,,,
        .BYTE   /1B,/C0,/01,/1E,/C4,,,
        .BYTE   ,/E0,/C1,/13,,/02,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,/B0,/C0,/13,,/02,,
        .BYTE   ,,,/1E,,,,
!
!
!
!
MPGM7:  .BYTE   /0A,/C0,/10,/1E,,,,
        .BYTE   /0A,/C0,/10,/1E,,,,
        .BYTE   /0B,/C0,/20,/1E,,,,
        .BYTE   /0B,/C0,/20,/1E,,,,
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /0C,/C0,/30,/1E,,,,
        .BYTE   /03,/90,,/1E,/44,/0E,,
        .BYTE   /02,/90,,/1E,/45,/49,,
        .BYTE   /01,/90,,/1E,/47,/EF,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,,,/83,,/13,,
        .BYTE   ,,,/1E,,,,
        .BYTE   ,,,/03,,/15,,
        .BYTE   ,,,/1E,,,,
```

```
.BYTE   /1B,/C0,/01,/1E,/C4,,,
.BYTE   ,/E0,/41,/13,,/16,,
.BYTE   ,,,/1E,,,,
.BYTE   ,/A0,/40,/13,,/16,,
.BYTE   ,,,/1E,,,,
.BYTE   ,/B0,/40,/1E,,,,
.BYTE   /03,/90,,/1E,/44,/23,,
.BYTE   /01,/90,,/1E,/45,/94,,
.BYTE   /02,/90,,/1E,/47,/FF,,
.BYTE   ,,,/1E,,,,
.BYTE   ,,,/83,,/21,,
.BYTE   ,,,/1E,,,,
.BYTE   ,,,/03,,/23,,
.BYTE   ,,,/1E,,,,
.BYTE   /1B,/C0,/01,/1E,/C4,,,
.BYTE   ,/E0,/51,/13,,/24,,
.BYTE   ,,,/1E,,,,
.BYTE   ,/A0,/50,/13,,/24,,
.BYTE   ,,,/1E,,,,
.BYTE   ,/B0,/50,/1E,,,,
.BYTE   /01,/90,,/1E,/44,/18,,
.BYTE   /02,/90,,/1E,/45,/2F,,
.BYTE   /03,/90,,/1E,/45,/51,,
.BYTE   ,,,/1E,,,,
.BYTE   ,,,/03,,/2D,,
.BYTE   ,,,/1E,,,,
.BYTE   /1B,/C0,/01,/1E,/C4,,,
.BYTE   ,/E0,/61,/13,,/2E,,
.BYTE   ,,,/1E,,,,
.BYTE   ,/B0,/60,/1E,,,,
.BYTE   /04,/50,,/1E,/86,,,
.BYTE   ,,,/43,,/36,,
.BYTE   ,,,/1E,,,,
.BYTE   /04,/60,,/1E,/86,,,
.BYTE   ,,,/43,,/3B,,
.BYTE   ,,,/1E,,,,
.BYTE   /04,/07,,/13,,/3C,,
.BYTE   ,,,/1E,,,,
.BYTE   /05,/60,,/1E,/86,,,
.BYTE   ,,,/43,,/3B,,
.BYTE   ,,,/1E,,,,
.BYTE   /05,/07,,/13,,/3C,,
.BYTE   ,,,/1E,,,,
.BYTE   /06,/07,,/1E,,,,
.BYTE   /08,/70,,/1E,/C1,,,
.BYTE   ,/E0,/C1,/13,,/02,,
.BYTE   ,,,/1E,,,,
```

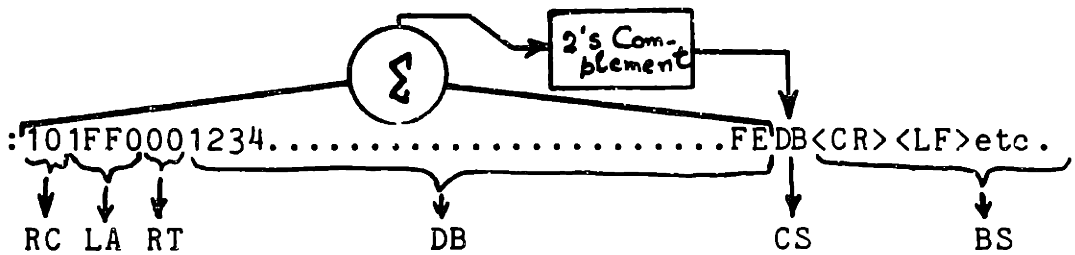
```
!
!
!
!
!
!   CONTROL BLOCK DESCRIPTION
!

    .=/1FE0            !Stack grows downwards from here
```

```
SAVAF:  .BLKB   2       ;save area for AF
SAVDE:  .BLKB   2       ;save area for DE
SAVBC:  .BLKB   2       ;save area for BC
SAVHL:  .BLKB   2       ;save area for HL
STACK:
SAVPC:  .BLKB   2       ;save area for program counter
SAVSP:  .BLKB   2       ;save area for Stack Pointer
RAMPTR: .BLKB   2       ;save area for RAM support pointer
BRKA:   .BLKB   2       ;save area for breakpoint address
BRKD:   .BLKB   1       ;save area for breakpoint depth
DSPLM:  .BLKB   1       ;display mode
                        ;   0 - indicates HL pointed format
                        ;   other values indicate REG format
RUNM:   .BLKB   1       ;run mode
                        ;   0 - free running
                        ;   other - single step(BKPT on M1)
PANBOX: .BLKB   8       ;miscellaneous buffer
RPVECT: .BLKB   1       ;save area for RTP vector
RPBRKD: .BLKB   1       ;save area for RTP breakpoint depth
RPSTRT: .BLKB   1       ;save area for default start address
ERCODE: .BLKB   1       ;Error Code
STATUS: .BLKB   1       ;status as follows
                        ;   bit7=1 - ICM/ICT initialized
                        ;   bit3 thru bit0 = S7,S6,S1,S0
;
;
        .END
```

APPENDIX - G.1

Intel's Hex-Format



RC -> # of data bytes in record (2 digits).
Ranges from 0H to 10H.
Must be 0 on last record.

LA -> Load Address (4 digits).

RT -> Record Type (2 digits).
0 = Normal
1 = EOF ; RC = 0, LA = Execution Address of program.

[Note: Last record may also be a 0-length normal record if no Execution Address exists.]

DB -> Data Bytes (2 digits each).

CS -> Checksum (2 digits).
[Mod_256 sum of each byte from the RC to the last data byte inclusive.]

BS -> Other garbage. Ignored. Nothing matters until the next ":" .

```
#include <stdio.h>
#define  zero   0
#define  fixcnt 16
#define  colon  58
#define  one    1
#define  twofs  255
main()   /* This program converts .dld format files into
            Intel-hex format absolute load modules, suitable
            for burning PROMs on MDS system. */
         {
            int badrhi,badrlo,bytcnt,c,chsum,count;
            int filler = 0;
start:   badrlo = getchar();
         badrhi = getchar();
         bytcnt = getchar();
         if (bytcnt != 0 )
            { while ( bytcnt > 0 )
                 {
                 count = ( bytcnt < fixcnt ) ? bytcnt:fixcnt;
                 chsum = badrhi + badrlo + count;
                 printf( "%1c", colon );
                 printf( "%02x", count );
                 printf( "%02x", badrhi );
                 printf( "%02x", badrlo );
                 printf( "%02x", zero );
                 while ( count-- > 0 )
                         {
                         c = getchar();
                         chsum += c ;
                         printf( "%02x", c );
                         --bytcnt;
                         };
                 chsum = - chsum;
                 chsum &= 255;
                 printf( "%02x", chsum );
                 printf( "0 );
                 if( ( badrlo += 16 ) > 255 )
                         {  badrlo -= 256;
                            ++badrhi;
                         };
                 };
             goto start;
            }
         else
            {
             printf( "%1c", colon );
             while ( filler++ < 3 )
```

```
        printf( "%02x", zero );
    printf( "%02x", one );
    printf( "%2x", twofs );
   };
}
```

APPENDIX - H

```
#include <stdio.h>

#define unhex(c) ('0' <= c && c <= '9' ? c-'0':c-'A'+10)

main(argc,argv)
        int argc;
        char **argv;
{
int done;
FILE *file;
done = 0;
while(--argc>0)
        {
        done = 1;
        if((file = fopen(*++argv, "r"))==NULL)
                {
                printf("unhex: can't open %s0, *argv);
                continue;
                }
        filter(file);
        }
if(!done)
filter(stdin);
return(0);
}

filter(file)
register FILE *file;
{
register int c, t;
while ((c = getc(file)) >= 0)
        {
        if (c == ' n' || c == ' r' || c == ' t' || c == ' ')
        continue;
        t = getc(file);
        c = (unhex(c) << 4) + unhex(t);
        putchar(c);
        }
}
```

APPENDIX - J.1

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A 90 00 1E C4 00 00 00	02	SHWL	IR1,#00
8018	1E E0 01 1E C4 00 00 00	03	SHWL	ROSA,ROSB
8020	1E E0 01 1E C4 00 00 00	04	SHWL	ROSA,ROSB
8028	1E E0 01 1E C4 00 00 00	05	SHWL	ROSA,ROSB
8030	1E E0 01 1E C4 00 00 00	06	SHWL	ROSA,ROSB
8038	1E E0 01 1E C4 00 00 00	07	SHWL	ROSA,ROSB
8040	1E E0 01 1E C4 00 00 00	08	SHWL	ROSA,ROSB
8048	1E E0 01 1E C4 00 00 00	09	SHWL	ROSA,ROSB
8050	00 E0 C1 13 00 02 00 00	0A	MOVB	ROSB,OR3
			BR	#02
8058	00 00 00 1E 00 00 00 00	0B	NOP	

APPENDIX - J.2

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A 01 00 1E 00 00 00 00	02	MOVA	IR1,GR1
8018	0B 10 00 1E 99 00 00 00	03	ADD	IR2,GR1
8020	00 00 00 A3 00 08 00 00	04	BAC	#08
8028	00 00 00 1E 00 00 00 00	05	NOP	
8030	00 A0 C1 13 00 02 00 00	06	MOVB	ALU,OR3
			BR	#02
8038	00 00 00 1E 00 00 00 00	07	NOP	
8040	00 B0 C0 13 00 02 00 00	08	MOVB	LIMHI,OR3
			BR	#02
8048	00 00 00 1E 00 00 00 00	09	NOP	

APPENDIX - J.3

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A 01 00 1E 00 00 00 00	02	MOVA	IR1,GR1
8018	0B 10 00 1E 44 00 00 00	03	MLIR	IR2,GR1
8020	0C 90 00 1E 47 01 00 00	04	MSIR	IR3,#01
8028	00 00 00 1E 00 00 00 00	05	NOP	
8030	00 B0 C1 13 00 02 00 00	06	MOVB	MACL,OR3
			BR	#02
8038	00 00 00 1E 00 00 00 00	07	NOP	

APPENDIX - J.4

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A 04 00 1E 00 00 00 00	02	MOVA	IR1,GR4
8018	0B 05 00 1E 00 00 00 00	03	MOVA	IR2,GR5
8020	0C 06 00 1E 00 00 00 00	04	MOVA	IR3,GR6
8028	04 50 00 1E 86 00 00 00	05	CMPR	GR4,GR5
8030	00 00 00 43 00 0D 00 00	06	BAHC	#0D
8038	00 00 00 1E 00 00 00 00	07	NOP	
8040	04 60 00 1E 86 00 00 00	08	CMPR	GR4,GR6
8048	00 00 00 43 00 12 00 00	09	BAHC	#12
8050	00 00 00 1E 00 00 00 00	0A	NOP	
8058	00 40 C0 13 00 02 00 00	0B	MOVB	GR4,OR3
			BR	#02
8060	00 00 00 1E 00 00 00 00	0C	NOP	
8068	05 60 00 1E 86 00 00 00	0D	CMPR	GR5,GR6

8070	00 00 00 43 00 12 00 00	0E	BAHC	#12
8078	00 00 00 1E 00 00 00 00	0F	NOP	
8080	00 50 C0 13 00 02 00 00	10	MOVB	GR5,OR3
			BR	#02
8088	00 00 00 1E 00 00 00 00	11	NOP	
8090	00 60 C0 13 00 02 00 00	12	MOVB	GR6,OR3
8098	00 00 00 1E 00 00 00 00	13	NOP	

APPENDIX - J.5

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A C0 10 1E 00 00 00 00	02	MOVN	IR1,GR1
8018	0A C0 10 1E 00 00 00 00	03	MOVN	IR1,GR1
8020	0B C0 20 1E 00 00 00 00	04	MOVN	IR2,GR2
8028	0B C0 20 1E 00 00 00 00	05	MOVN	IR2,GR2
8030	0C C0 30 1E 00 00 00 00	06	MOVN	IR3,GR3
8038	0C C0 30 1E 00 00 00 00	07	MOVN	IR3,GR3
8040	03 90 00 1E 44 0E 00 00	08	MLIR	GR3,#0E
8048	02 90 00 1E 45 49 00 00	09	MAIR	GR2,#49
8050	01 90 00 1E 47 EF 00 00	0A	MSIR	GR1,#EF
8058	00 00 00 1E 00 00 00 00	0B	NOP	
8060	00 00 00 83 00 13 00 00	0C	BMXP	#13
8068	00 00 00 1E 00 00 00 00	0D	NOP	
8070	00 00 00 03 00 15 00 00	0E	BM15	#15

```
8078  00 00 00 1E 00 00 00 00   0F   NOP

8080  1B C0 01 1E C4 00 00 00   10   SHWL    MACL,MACH

8088  00 E0 C1 13 00 02 00 00   11   MOVB    ROSB,OR3
                                     BR      #02

8090  00 00 00 1E 00 00 00 00   12   NOP

8098  00 A0 C0 13 00 02 00 00   13   MOVB    LIMLO,OR3
                                     BR      #02

80A0  00 00 00 1E 00 00 00 00   14   NOP

80A8  00 B0 C0 13 00 02 00 00   15   MOVB    LIMHI,OR3
                                     BR      #02

80B0  00 00 00 1E 00 00 00 00   16   NOP
```

APPENDIX - J.6

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A C0 10 1E 00 00 00 00	02	MOVN	IR1,GR1
8018	0A C0 10 1E 00 00 00 00	03	MOVN	IR1,GR1
8020	0B C0 20 1E 00 00 00 00	04	MOVN	IR2,GR2
8028	0B C0 20 1E 00 00 00 00	05	MOVN	IR2,GR2
8030	0C C0 30 1E 00 00 00 00	06	MOVN	IR3,GR3
8038	0C C0 30 1E 00 00 00 00	07	MOVN	IR3,GR3
8040	03 90 00 1E 44 23 00 00	08	MLIR	GR3,#23
8048	01 90 00 1E 45 94 00 00	09	MAIR	GR1,#94
8050	02 90 00 1E 47 FF 00 00	0A	MSIR	GR2,#FF
8058	00 00 00 1E 00 00 00 00	0B	NOP	
8060	00 00 00 83 00 13 00 00	0C	BMXP	#13
8068	00 00 00 1E 00 00 00 00	0D	NOP	
8070	00 00 00 03 00 15 00 00	0E	BM15	#15

8078	00 00 00 1E 00 00 00 00	OF	NOP	
8080	1B C0 01 1E C4 00 00 00	10	SHWL	MACL,MACH
8088	00 E0 C1 13 00 02 00 00	11	MOVB	ROSB,OR3
			BR	#02
8090	00 00 00 1E 00 00 00 00	12	NOP	
8098	00 A0 C0 13 00 02 00 00	13	MOVB	LIMLO,OR3
			BR	#02
80A0	00 00 00 1E 00 00 00 00	14	NOP	
80AA	00 B0 C0 13 00 02 00 00	15	MOVB	LIMHI,OR3
			BR	#02
80B0	00 00 00 1E 00 00 00 00	16	NOP	

APPENDIX - J.7

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A C0 10 1E 00 00 00 00	02	MOVN	IR1,GR1
8018	0A C0 10 1E 00 00 00 00	03	MOVN	IR1,GR1
8020	0B C0 20 1E 00 00 00 00	04	MOVN	IR2,GR2
8028	0B C0 20 1E 00 00 00 00	05	MOVN	IR2,GR2
8030	0C C0 30 1E 00 00 00 00	06	MOVN	IR3,GR3
8038	0C C0 30 1E 00 00 00 00	07	MOVN	IR3,GR3
8040	01 90 00 1E 44 18 00 00	08	MLIR	GR1,#18
8048	02 90 00 1E 45 2F 00 00	09	MAIR	GR2,#2F
8050	03 90 00 1E 45 51 00 00	0A	MAIR	GR3,#51
8058	00 00 00 1E 00 00 00 00	0B	NOP	
8060	00 00 00 03 00 11 00 00	0C	BM15	#11
8068	00 00 00 1E 00 00 00 00	0D	NOP	

```
8070  1B C0 01 1E C4 00 00 00   0E   SHWL   MACL,MACH

8078  00 E0 C1 13 00 02 00 00   0F   MOVB   ROSB,OR3
                                     BR     #02

8080  00 00 00 1E 00 00 00 00   10   NOP

8088  00 B0 C0 13 00 02 00 00   11   MOVB   LIMHI,OR3
                                     BR     #02

8090  00 00 00 1E 00 00 00 00        NOP
```

APPENDIX - J.8

MGR ADDR	MICRO_CODE	MPGM ADDR	ASSEMBLY CODE	
8010	0A C0 10 1E 00 00 00 00	02	MOVN	IR1,GR1
8018	0A C0 10 1E 00 00 00 00	03	MOVN	IR1,GR1
8020	0B C0 20 1E 00 00 00 00	04	MOVN	IR2,GR2
8028	0B C0 20 1E 00 00 00 00	05	MOVN	IR2,GR2
8030	0C C0 30 1E 00 00 00 00	06	MOVN	IR3,GR3
8038	0C C0 30 1E 00 00 00 00	07	MOVN	IR3,GR3
8040	03 90 00 1E 44 0E 00 00	08	MLIR	GR3,#0E
8048	02 90 00 1E 45 49 00 00	09	MAIR	GR2,#49
8050	01 90 00 1E 47 EF 00 00	0A	MSIR	GR1,#EF
8058	00 00 00 1E 00 00 00 00	0B	NOP	
8060	00 00 00 83 00 13 00 00	0C	BMXP	#13
8068	00 00 00 1E 00 00 00 00	0D	NOP	
8070	00 00 00 03 00 15 00 00	0E	BM15	#15

8078	00 00 00 1E 00 00 00 00	0F	NOP	
8080	1B C0 01 1E C4 00 00 00	10	SHWL	MACL,MACH
8088	00 E0 41 13 00 16 00 00	11	MOVB	ROSB,GR4
			BR	#16
8090	00 00 00 1E 00 00 00 00	12	NOP	
8098	00 00 40 13 00 16 00 00	13	MOVB	LIMLO,GR4
			BR	#16
80A0	00 00 00 1E 00 00 00 00	14	NOP	
80A8	00 B0 40 1E 00 00 00 00	15	MOVB	LIMHI,GR4
80B0	03 90 00 1E 44 23 00 00	16	MLIR	GR3,#23
80B8	01 90 00 1E 45 94 00 00	17	MAIR	GR1,#94
80C0	02 90 00 1E 47 FF 00 00	18	MSIR	GR2,#FF
80C8	00 00 00 1E 00 00 00 00	19	NOP	
80D0	00 00 00 83 00 21 00 00	1A	BMXP	#21
80D8	00 00 00 1E 00 00 00 00	1B	NOP	
80E0	00 00 00 03 00 23 00 00	1C	BM15	#23
80E8	00 00 00 1E 00 00 00 00	1D	NOP	
80F0	1B C0 01 1E C4 00 00 00	1E	SHWL	MACL,MACH
80F8	00 E0 51 13 00 24 00 00	1F	MOVB	ROSB,GR5

app-J.8

```
                                                  BR      #24

8100  00 00 00 1E 00 00 00 00    20      NOP

8108  00 A0 50 13 00 24 00 00    21      MOVB    LIMLO,GR5
                                         BR      #24

8110  00 00 00 1E 00 00 00 00    22      NOP

8118  00 B0 50 1E 00 00 00 00    23      MOVB    LIMHI,GR5

8120  01 90 00 1E 44 1E 00 00    24      MLIR    GR1,#18

8128  02 90 00 1E 45 2F 00 00    25      MAIR    GR2,#2F

8130  03 90 00 1E 45 51 00 00    26      MAIR    GR3,#51

8138  00 00 00 1E 00 00 00 00    27      NOP

8140  00 00 00 03 00 2D 00 00    28      BM15    #2D

8148  00 00 00 1E 00 00 00 00    29      NOP

8150  1B C0 01 1E C4 00 00 00    2A      SHWL    MACL,MACH

8158  00 E0 61 13 00 2E 00 00    2B      MOVB    ROSB,GR6
                                         BR      #2E

8160  00 00 00 1E 00 00 00 00    2C      NOP

8168  00 B0 60 1E 00 00 00 00    2D      MOVB    LIMHI,GR6

8170  04 50 00 1E 86 00 00 00    2E      CMPR    GR4,GR5

8178  00 00 00 43 00 36 00 00    2F      BAHC    #36
```

app - I.9

```
8180  00 00 00 1E 00 00 00 00   30   NOP
8188  04 60 00 1E 86 00 00 00   31   CMPR   GR4,GR6
8190  00 00 00 43 00 3B 00 00   32   BAHC   #3B
8198  00 00 00 1E 00 00 00 00   33   NOP
81A0  04 07 00 13 00 3C 00 00   34   MOVA   GR4,GR7
                                     BR     #3C
81A8  00 00 00 1E 00 00 00 00   35   NOP
81B0  05 60 00 1E 86 00 00 00   36   CMPR   GR5,GR6
81B8  00 00 00 43 00 3B 00 00   37   BAHC   #3B
81C0  00 00 00 1E 00 00 00 00   38   NOP
81C8  05 07 00 13 00 3C 00 00   39   MOVA   GR5,GR7
                                     BR     #3C
81D0  00 00 00 1E 00 00 00 00   3A   NOP
81D8  06 07 00 1E 00 00 00 00   3B   MOVA   GR6,GR7
81E0  08 70 00 1E C1 00 00 00   3C   SHBR   GR8,GR7
81E8  00 E0 C1 13 00 02 00 00   3D   MOVB   ROSB,OR3
                                     BR     #02
81F0  00 00 00 1E 00 00 00 00   3E   NOP
```

APPENDIX - K

RTP INSTRUCTION SET SUMMARY

The general syntax of the MICRo-ASSembler language statement is as follows;

```
LABEL:    OP-CODE           OPRND-A,OPRND-B,OPRND-C       ;COMMENT
(opt.)  +QUALIFIER(opt.)                  (sp. case)      (opt.)
```

1. Label Field

This field is optional and same rules apply regarding its use, as in case of conventional assembly language programming. Each symbol must be terminated with a colon(:) to form a valid label field.

```
Examples;       START:
                FOO: BAR:
                7$:
```

2. Op-Code Field

This field refers to an instruction mnemonic or assembler directive. In addition, op-codes may be suffixed with qualifier characters to denote special hardware functions. All op-codes are upto 4-character long. The 5th (and 6th) character, if present, always denotes the qualifier.

Examples; MOVN+

SDBL&

BAM

MOVB^

At the ehd, this Appendix lists all valid op-codes groupwise, including assembler directives.

2.1. Qualifiers

The qualifiers denote special function by hardware such as masking, additional destination operands, program cycle termination etc. All valid qualifier characters and their meaning are as follows;

"+" - denotes additional destination, to be specified by operand-C.

"&" - denotes masking on both buses to be turned on.

"&A" - denotes masking on bus-A only to be turned on.

"&B" - denotes masking on bus-B only to be turned on.

"^" - denotes program cycle termination.

3. Operand Field

app-K

The operand fields are seperated by comma(,) and are interpreted according to the operation specified by the Op-code field. If the specified operation requires use of a single bus only (as in MOVA or MOVB instructions), the first field specifies the source operand and the second field specifies the destination operand. Else, whenever both buses must be used, the first field specifies the bus-A operand and the second field specifies the bus-B operand. If the op-code is qualified with a "+" character, a third field must be stated, specifying the additional destination operand. At the end, all valid operands are listed. As shown there, some operands have limited communication capabilities.

An Immediate operand is prefixed with "#" character. Presently, all immediate operands must be specified by 2 hexadecimal characters.

Examples; IR3
GR1
MACL
#2F
ALU

4. Comment Field

The comment field must begin with a semi-colon(;) character and is optional. Same rules apply regarding its use as in case of conventional assembly language program.

LIST OF VALID RTP OP-CODES app-K

m/c code	op-code	action

(A) DATA-TRANSFER GROUP

m/c code	op-code	action
	MOVA	- Move the contents of the 1st operand to the 2nd operand using bus-A.
	MOVB	- Move the contents of the 1st operand to the 2nd operand using bus-B.

SPECIAL INSTRUCTIONS:

m/c code	op-code	action
	MAPN	- Map the contents of operand-A non-linearly and move result to operand-B.
	MUCR	- Map the contents of operand-A for Under-Color-Removal and move result to opr-B.

(B) EXECUTE GROUP

m/c code	op-code	action
99	ADD	- Add operand-A to operand-B, result in ALU.
86	SUBT	- Subtract opr-B from opr-A, result in ALU.
86	CMPR	- Compare operand-A with operand-B, result in CCR.
9F	DECR	- Decrement operand-A, result in ALU.
80	INCR	- Increment operand-A, result in ALU.
90	PASS	- Pass operand-A as result in ALU.

AB AND - Logically AND opr-A with opr-B, result in ALU.

A4 NAND - Logically NAND opr-A with opr-B, result in ALU.

AE OR - Logically OR opr-A with opr-B, result in ALU.

A1 NOR - Logically NOR opr-A with opr-B, result in ALU.

A6 EOR - Logically EOR opr-A with opr-B, result in ALU.

A9 NOR - Logically ENOR opr-A with opr-B, result in ALU.

83 LIML - Set ALU result to all zeros.

93 LIMH - Set ALU result to all ones.

40 MLIT - Multiply Integer and Truncate.

44 MLIR - Multiply Integer and Round-off.

48 MLTT - Multiply 2's-Complement and Truncate.

4C MLTR - Multiply 2's-Complement and Round-off.

41 MAIT - Multiply-accumulate Integer and Truncate.

45 MAIR - Multiply-accumulate Integer and Round-off.

49 MATT - Multiply-accumulate 2's-Complement and Truncate.

4D MATR - Multiply-accumulate 2's-C and Round-off.

43 MSIT - Multiply-subtract Integer and Truncate.

app-K

47 MSIR - Multiply-subtract Integer and Round-off.

4B MSTT - Multiply-subtract 2's-Complement and Truncate.

4F MSTR - Multiply-subtract 2's-Complement and Round-off.

64 MPL - Pre-load MAC TP .

62 MMPL - Pre-load MAC MSP .

61 MLPL - Pre-load MAC LSP .

67 MPEL - Pre-load MAC completely.

C0 SHBL - Shift byte left.

C1 SHBR - Shift byte right.

C4 SHWL - Shift word left.

C5 SHWR - Shift word right.

C2 ROBL - Rotate byte left.

C3 ROBR - Rotate byte right.

C6 ROWL - Rotate word left.

C7 ROWR - Rotate word right.

(C) BRANCH GROUP

03 BAZ - Branch on ALU zero.

23 BAM - Branch on ALU minus.

63 BALC - Branch on ALU logical carry.

43 BAHC - Branch on ALU hardware carry.

83 BAQ - Branch on ALU equal.

A3 BAV - Branch on ALU overflow.

A3 BM14 - Branch on MAC bit#14 high.

03 BM15 - Branch on MAC bit#15 high.

23 BM16 - Branch on MAC bit#16 high.

43 BM17 - Branch on MAC bit#17 high.

63 BM18 - Branch on MAC bit#18 high.

83 BMP - Branch on MAC TP minus.

C3 BNWD - Branch on new data arrival.

E3 BCON - Branch on Controller signal.

13 BR - Unconditional branch.

(D) SPECIAL GROUP

1E NOP - No operation.

(E) ASSEMBLER DIRECTIVES

app-K

START - Program origin, Location Counter = 02

END - End of program.

LIST OF VALID RTP OPERANDS

Operands		Description
GR1, GR2, GR3, GR4, GR5, GR6, GR7, GR8	}	Can talk and listen on both Bus-A and Bus-B
IDR, CCR, ALU, MACL, MACH	}	Can talk on both Bus-A and Bus-B ALU, MAC, ROSH always listen
ROSA	-	Can talk on Bus-A only
ROSB	-	Can talk on Bus-B only
IR1, IR2, IR3, ICT1, ICT2, ICT3, ICT4, ICT5, ICT6, ICT7	}	Can talk only on Bus-A only
OR1, OR2, OR3	-	Can listen only on Bus-B only
MAR1, MAR2, MAR3, MBRA, MMPL	}	Can eavesdrop only on Bus-A
MBRB, MLPL, MXPL, PGR	-	Can eavesdrop only on Bus-B

APPENDIX - L

```
   ;This version should be assembled using the
   ;two-pass MICROASSembler.
   ;
   ;This microprogram computes printing ink density YELLOW
   ;from input color R-G-B. Following assumtion is made as
   ;regards the state of the input to Real-time Processor,
   ;at the time of the activation of this program:
   ;               IR1 - contains RED
   ;               IR2 - contains GRN
   ;               IR3 - contains BLU
   ;
   ;
   ;
10$:     START                      ;origin program at PC=02
         MOVN     IR1,GR1           ;compute DR and save
         MOVN     IR1,GR2
         MOVN     IR2,GR2           ;compute DG and save
         MOVN     IR2,GR2
         MOVN     IR3,GR3           ;compute DB and save
         MOVN     IR3,GR3
   ;
   ;AT THIS POINT, DR-DG-DB VECTOR HAS BEEN COMPUTED
   ;STARTING FROM INPUT COLOR VECTOR R-G-B.
   ;
         MLIR     GR3,#0E           ;compute p.DB
         MAIR     GR2,#49           ;compute n.DG and accumulate
         MSIR     GR1,#EF           ;compute m.DR & subtract cum.
         NOP
         BMXP     11$               ;if underflow, clamp to zero
         NOP
         BM15     12$               ;if overflow, clamp to '/FF'
         NOP
         SHWL     MACL,MACH         ;shift MAC o/p left, word mode
   ;shift takes care of m,n,p 's format I.FFFFFFF
         MOVB     ROSB,GR4          ;save Y-bar
         BR       13$               ;continue
         NOP
11$:     MOVB     LIMLO,GR4         ;underflow, clamp value to zero
         BR       13$               ;continue
         NOP
12$:     MOVB     LIMHI,GR4         ;overflow, clamp value to 255
13$:     MLIR     GR3,#23           ;compute t.DB
         MAIR     GR1,#94           ;compute r.DG and accumulate
         MSIR     GR2,#FF           ;compute q.DR & subtract cum.
         NOP
         BMXP     21$               ;if underflow, clamp to zero
         NOP
```

```
        BM15      22$                 ;if overflow, clamp to '/FF'
        NOP
        SHWL      MACL,MACH           ;shift MAC o/p left, word mode
   ;shift is required for the same reason as above
        MOVB      ROSB,GR5            ;save C-bar
        BR        23$                 ;continue
        NOP
21$:    MOVB      LIMLO,GR5           ;underflow, clamp value to zero
        BR        23$                 ;continue
        NOP
22$:    MOVB      LIMHI,GR5           ;overflow, clamp value to 255
23$:    MLIR      GR1,#18             ;compute x.DR
        MAIR      GR2,#2F             ;compute y.DG and accumulate
        MAIR      GR3,#51             ;compute z.DB and accumulate
        NOP
        BMXP      31$                 ;if overflow, clamp to '/FF'
        NOP
        SHWL      MACL,MACH           ;shift MAC o/p left, word mode
   ;shift is required for the same reason as above
        MOVB      ROSB,GR6            ;save M-bar
        BR        32$                 ;continue
        NOP
31$:    MOVB      LIMHI,GR6           ;overflow, clamp value to 255
   ;
   ;AT THIS POINT, INK DENSITY APPROXIMATIONS Y-BAR, C-BAR,
   ;M-BAR HAVE BEEN COMPUTED.
   ;
   ;FOLLOWING PROGRAM SEGMENT COMPUTES 3_DIMENSIONAL LINEAR
   ;INTERPOLATION USING INK_CORRECTION_TABLE.
   ;
32$:    SHBL&     GR4,GR5             ;left shift Y_bar,C_bar mask on
        SHBL      ROSA,ROSB           ;one more time
        MOVA      ROSA,GR1            ;save [Y]8
        MOVB      ROSB,GR2            ;save [C]8
   ;format for [T]8 is 2's C 0.FFFFFFF , F=0 or 1
        SHBL&     GR6                 ;left shift M-bar with mask on
        SHBL      ROSA                ;one more time
        MOVA      ROSA,GR3            ;save [M]8
        MLTR      GR1,GR2             ;compute [Y]8.[C]8
        NOP
        SHWL      MACL,MACH           ;shift MAC o/p left, word mode
   ;shift justifies decimal point
        MOVB      ROSB,GR7            ;save [Y]8.[C]8
        MLTR      GR2,GR3             ;compute [C]8.[M]8
        NOP
        SHWL      MACL,MACH           ;shift MAC o/p left, word mode
        MOVB      ROSB,GR8            ;save [C]8.[M]8
        MLTR      GR1,GR3             ;compute [M]8.[Y]8
        NOP
        SHWL      MACL,MACH           ;shift MAC o/p left, word mode
        MLTR      ICT7,ROSB           ;compute
        MATR      ICT6,GR8            ;commpute
        MATR      ICT5,GR7            ;compute
        MATR      ICT4,GR3            ;compute
```

```
        MATR    ICT3,GR2        ;compute
        MATR    ICT2,GR1        ;compute
        MATR    ICT1,#EF        ;add Y'
        NOP
        BMXP    41$             ;check if underflow
        NOP
        BM15    42$             ;check if overflow
        NOP
        SHWL    MACL,MACH       ;align result
        MOVB^   ROSB,OR3        ;output result & terminate
        BR      51$
        NOP
41$:    MOVB^   LIMLO,OR3       ;underflow, output 0, terminate
        BR      51$
        NOP
42$:    MOVB^   LIMHI,OR3       ;overflow, output /FF, terminate
51$     BNW     10$             ;if new data, start new cycle
        NOP
        BR      51$             ;else, loop & check again
        NOP
        END
```