To the Graduate Council:

I am submitting herewith a thesis written by James M. Ross, III entitled "A Music Transcription System for the Apple II Microcomputer." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

J. Michael Moshell, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

A MUSIC TRANSCRIPTION SYSTEM FOR THE APPLE II MICROCOMPUTER

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

James M. Ross, III

December 1980

3046729

ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to his parents for their continued patience and support during his extended scholastic career.

ABSTRACT

The APPLE II microcomputer currently has excellent musical I/O capabilities. The ALF Music Synthesizer card supplies musical output, and the PAIA digital keyboard in conjunction with a home-made interface card supplies musical output. The user programs the ALF card by specifying the pitch and volume for either of three available voices (additional cards supply three more voices each). The PAIA hardware continually scans its keys, signalling during each scan when it finds a key pressed. The keyboard interface generates an interrupt request to the APPLE for each such PAIA signal.

This thesis supplies the software necessary to 1) turn the APPLE into an organ/synthesizer by accepting input from the PAIA keyboard interface and generating the appropriate signals to drive the ALF synthesizer card; generate data representing a "digital recording" of what has been played on the keyboard; and replay the song using this song data; and 2) transcribe the "live" song data into musical notation which will be shown using APPLE graphics; generate musical data compatible with a graphics editor; and save song data on a magnetic disk for later retrieval either to replay the song, notate it, or edit it.

The software is written in the Apple version of UCSD PASCAL. The first part, due to the real-time nature of the problem, is written using the PASCAL compatible 6502 assembler; the second part is implemented using the PASCAL language.

This software provides a powerful musical instrument--with the addition of the editor, for example, one can play an original song, save it immediately, and come back later to change it.

The scope of this thesis is currently being extended to provide

1. a graphics-oriented music text editor; the data generated is compatible with the song data generated by the live playing routines.
2. real-time playing commands which alter the generated sound, i.e., change the envelope parameters, volume, and tone of the music.

TABLE OF CONTENTS

CHAPTER		PAGE
I.	BACKGROUND	1
	The ALF Synthesizer Card	1
	The Apple-Paia Digital Keyboard Interface Card	3
II.	THE KEYBOARD PREPROCESSOR	4
	Description	4
	The Song Data	9
	Associated Routines	12
	How to Use the Keyboard Playing Routines	16
	Communication Between Pascal and the Lower Level Routines	18
III.	DESCRIPTION OF THE SYSTEM	22
IV.	TRANSCRIBING LIVE KEYBOARD INPUT	29
	Method	29
	Some Specific Solutions to Transcription Problems	40
V.	RESULTS	47
BIBLIOGRAPHY		50
APPENDICES		52
APPENDIX A.	ZERO PAGE USAGE: PASCAL VERSION	53
APPENDIX B.	THE KEYBOARD PREPROCESSOR: BASIC VERSION	54
APPENDIX C.	MEMORY ADDRESSES FOR KEYBOARD PREPROCESSOR: BASIC VERSION	59
VITA		60

CHAPTER I

BACKGROUND

1. THE ALF SYNTHESIZER CARD

Each ALF Synthesizer Card supplies three output channels, all of which may be active at the same time, thus enabling three simultaneous voices to be played. One of the channels may be used to enable pulse mode rather than normal, unpulsed mode. Using a channel in this way removes its capability as a voice output, so this work never enables pulse mode.

Each card has eight programmable input ports which are accessed by simply storing a value in the appropriate Apple I/O addresses. The correct address is a function of the Apple base I/O address, the slot in which the synthesizer resides, and which of the eight ports is desired. The ports are numbered 0 to 7. Two ports control the mode of output (normal or pulsed); two each of the remaining six control the pitch and volume of one of the three channels.

Storing a value in a port that controls the volume of a channel immediately affects the volume of that channel. Since each port is a single byte, a range of volumes from 0 (silence) to 255 (maximum volume) may be specified. Naturally these values are related to the volume setting of the amplifier to which the ALF card is connected. By

changing the volume level of a voice at the appropriate time, sound envelopes may be created. Documentation on how envelopes are created in this work may be found in the report of the independent study project to add this feature (Harry Delugauch, Summer 1980).

Storing a value in a port that controls the pitch of a channel immediately affects the pitch output by that channel. The values which must be sent are called DIVISORS. Divisors are so-called because they are a value by which the output of the pitch generator on the ALF card is divided to determine the desired pitch output. The pitch generator outputs a square wave at a frequency of about 1,782,000 cycles per second. The ALF card expects a sixteen bit value--it expects the low byte of this value first, followed by the high byte.

To obtain divisor values corresponding to piano notes, we use the formula

```
F(N) = F(0) * (2 ** (N / 12))
```

where F(0) is the frequency of the lowest note in an octave and N is the note (numbered 0 to 11) within that octave. The divisor we want is the value 1,782,000 divided by the value obtained by this formula (since dividing 1,782,000 by the resulting value will give us back the original frequency desired).

2. THE APPLE-PAIA DIGITAL KEYBOARD INTERFACE CARD

The interface card which connects the Apple and the Paia keyboard was designed and built by Doug Barnhardt as an independent study project at the University of Tennessee.

The Paia hardware continually scans the keys, generating signals when it finds a key pressed. The scan is from left to right. At the end of a complete scan of the keys, it generates an end of scan signal (whether or not a key was pressed). Each key and the end of scan have specific numbers associated with them. The thirty-six keys are numbered from $10 to $34 (decimal 16 to 52); the end of scan is numbered $3F (decimal 127). The interface card generates an interrupt request whenever any of these signals are generated by the Paia hardware. The number associated with the cause of the interrupt is displayed at a specific memory address (one of the I/O ports assigned to each slot of the Apple). This number is used by the keyboard software of this project to index into a table of divisors; this value is then sent to the ALF divisor port for one of the three channels.

The interface card thus tells us only when it finds a key pressed or when it reaches the end of a key scan. It does not tell us when a key that has been pressed is released. This is the responsibility of the software accessing the data offered by the interface card.

CHAPTER II

THE KEYBOARD PREPROCESSOR

Since the keyboard interface generates interrupts for every key pressed during one scan, we need a preprocessor which will sift through this information to find out what we really want to know: when is a key initially pressed, when is it released, and how much time elapses between these two events? When a key is pressed, we want to program the ALF card to output the appropriate pitch; when a key is released, we want to stop the pitch output.

1. DESCRIPTION

The action of the preprocessor is simple--it maintains a list of keys currently pressed; these correspond to the notes currently heard through the synthesizer card. This list is ordered in the same way as the keyboard scan (i.e., from low key to high), so we can maintain a pointer to the node of the next expected key, that is, the one we saw at this point during the last scan. If the key causing the interrupt matches the expected key, we simply move the expected key pointer forward--we saw this key pressed last time, which means we have already played it. If the interrupting key is less than the expected key, then the interrupting key has been pressed during the last scan and

we note it as such: we play the corresponding note through the synthesizer card and insert a new node before the one pointed to. In this case, the pointer remains where it is, since we still expect this key to come up next in the scan. If the interrupting key is greater than the expected key, then the expected key has been released during the last scan--we "unplay" it on the synthesizer card and remove its node from the list. In this case, we move the expected key pointer forward and again compare the interrupting key (during the same interrupt period; we continue comparisons until we decide what to do with the interrupting key).

The keyboard interface also generates interrupts for the end of scan, which occurs just after the highest key has been polled. We simply treat this as an extra note which is always on the list of expected keys. It is different from the other keys only in that it never causes a pitch to be generated. Furthermore, since the value of the end of scan "key" is the highest of the keys and since it is placed on the list initially, there is no danger of removing it from the list.

The main data structure here, then, is a doubly linked circular list of keys currently pressed. The head of the list points to the expected key and moves along the list depending on whether the interrupting key is equal to, less than, or greater than the key pointed to. Initially, the

expected key is the end of scan, and both front and back links point to this single node.

Having a circular list necessitates a free space data structure as well. This is maintained simply as a stack which initially contains as many pointers to nodes as there are available voices (i.e., three if there is a single synthesizer card being used, six if two, etc.). Pointers are popped as nodes are needed and pushed as they are released. This leads to the problem of what to do when more keys are pressed than there are voices, a condition signalled by stack underflow. Our solution is to ignore the extra keys until room is made by releasing currently pressed keys.

In short, the keyboard preprocessor sifts through the constant stream of information coming from the keyboard and decides when a key has changed state to being pressed or released. This refined information is used to play notes through the synthesizer card as well as to generate song data.

System Timing

The preprocessor counts the number of time units occurring between keyboard changes. A time unit is the amount of time needed to process a single interrupt and is held constant by using the analog paddle input as a clock. A time unit of approximately 1/100 second is obtained by

poking the paddle to life three times, waiting each time for it to die. "Poking the paddle to life" refers to how numbers are obtained from the paddle: one accesses a specific memory location associated with the paddle by a simple store operation. This in effect causes the paddle to be charged; one then counts how long it takes for the paddle to discharge (this is signalled by a bit being set in another specific memory location). Turning the paddle fully clockwise maximizes the time between the initial charge and the discharge. Each poke, then, gives us about 1/300 second (provided the paddle is turned fully clockwise) of waiting time, during which we can do other things. One waiting time is used by the keyboard routine itself. The second time is reserved for producing a metronome and reading the Apple keyboard. The third is reserved for envelope production. At the end of the third period, we poke the paddle to start things over again and then execute a return from interrupt, having done everything we needed to do. Since interrupts are guaranteed to be pending, as they happen much faster than 1/100 second, we have a consistent clocking method.

A zero page byte of timing information, called STOPWTCH, gets incremented during each interrupt when no changes on the keyboard have been noted by the preprocessor. It gets stored and reset to zero whenever a key is pressed or released, or whenever 255 time units have elapsed with no changes noted.

Metronome Timing

As noted above, timing is obtained via the analog input device (Paddle 0). By poking the correct memory location three times, we obtain a basic time unit of approximately 1/100 second. A major problem in music transcription is deciding the tempo of a played piece. We short circuit this problem by having the user specify ahead of time what the tempo of the piece will be, allowing the user and the system to synchronize their tempos. To prompt the user with the correct tempo, the system uses a metronome which is heard through the built-in Apple II speaker and which is seen on the console screen.

During the start of each beat (for the first six time units) the speaker is toggled and a single low-res pixel is shown. The result is a low-volume "click" and a flash on the screen.

The user specifies the tempo using standard Maelzel metronome units. This simply represents the number of beats per minute the metronome should click. This value gets modified to represent the number of basic time units per beat. The function to do this is

```
TIME UNITS = 60 / METRONOME * (TIME UNITS PER SECOND)
```

For example, if there are 100 time units per second, a Maelzel marking of 120 yields

60 / 120 * 100 = 50

Thus every 50 "ticks" of our 1/100 second stopwatch, i.e., every 1/2 second, we output a metronome click.

We use a single byte of information to represent time units per metronome click, so we can offer Maelzel values of 24 to 255 (24 yields 250 time units per beat; 255 yields 24).

2. THE SONG DATA

The song data starts at location $1600 (decimal 5632). The location of the last byte of song data is stored at page zero locations $E2 and $E3 (decimal 226 and 227) in the standard low byte-high byte fashion. The data is organized in two-byte units. The first byte is a command and the second is a time parameter showing the amount of time its associated command is "in effect." (We call the first byte a command because We think of it as telling me what to do as I look at it. You can also look at it as information about what happened--that is what it really is.) There are three kinds of commands:

1. KEYPRESS: this is generated when a key is first pressed.
2. KEYRELEASE: this is generated when a key is released.
3. WAIT: this is generated when the time parameter of the previous command will not hold the time value needed. The WAIT command is a kind of extension of the time parameter.

When one of these commands is inserted in the song data, the preprocessor counts the number of time units until the next command needs to be inserted, which is when a key has changed state. This value is the time parameter for this command. Since one byte can represent only 255 time units (about 2.5 seconds), if more than 255 time units elapse before a change is noted, the WAIT command is inserted which has a time parameter of its own.

A KEYPRESS command is the actual code value of the key pressed. Thus, for the Paia keyboard, KEYPRESS commands range from $10 to $34 (decimal 16 to 52), since these are the codes generated by the keyboard. In this way, KEYPRESS commands tell us not only that a key is pressed but also which specific key. A KEYRELEASE command is the same as a KEYPRESS, but with a 1 in the most significant bit (making it negative or greater than 127, depending on how you look

at it). Thus KEYRELEASE commands range from $90 to $B4 (decimal 144 to 180). You need only subtract 128 from a KEYRELEASE command to obtain the key code it's talking about. The end of song command is the value $00, since this is the code generated by the <SF> button on the keyboard. A WAIT command is $FF (decimal 255).

Two major factors led to our choosing this form of song data:

1. It places no limitation on the number of voices available, and, furthermore, the number of voices available place no limitation on the song data. An alternative form of song data would have had a section of data representing keys played on voice 1, another for voice 2, and so on. The method chosen needs no special set-up to be used for any number of voices.
2. The song data can be parsed as a left-to-right stream of tokens and could be easily parsed using any of the standard parsing techniques. As it stands, the song data is a simple LR 0 grammar.

3. ASSOCIATED ROUTINES

Several other routines are associated with the keyboard preprocessor. Two, INITPROG and INITSONG, are used to set up the preprocessor; another, PLAYBACK, uses song data generated by the preprocessor to replay the song represented. These routines are desribed below.

The PLAYBACK Routine

The PLAYBACK routine simply uses the stored song data to generate "pseudo-interrupts" representing keypresses and keyreleases. When it sees a keypress command, it plays the appropriate note (after using the insert routine of KEYBOARD to find an available voice); when it sees a keyrelease command it turns down the volume of the appropriate channel by looking on the list of currently playing notes, finding the position of the node holding the value of the note to release, and using the node number as the voice number to turn down.

When it sees a wait parameter, it sends the value to the WAIT routine of procedure KEYBOARD after turning the value into two's complement form (since WAIT counts up to $FF before returning with a flag that the current time period has expired). This insures that the same time elapses between musical "events" as did during actual live keyboard playing. WAIT simply counts one time unit by

successively poking the game paddle. During the second and third time periods (a time unit is made up of three time periods), routine WAIT gathers input from the Apple console, outputs a metronome, and produces volume envelopes. PLAYBACK then loops, calling WAIT until all the time units are taken care of, then picks up the next command. This continues until the end of song command is found which causes the routine to return to the Pascal host.

When calling INSERT to play a note and insert it into the list of currently playing notes, PLAYBACK always asks that the note be inserted before the last node (the end of scan node) in the list. PLAYBACK does not depend on an ordered list as does KEYBOARD (this is so the song data, even when edited, is compatible with the PLAYBACK routine). But we saw no reason to build another list handling routine for PLAYBACK, so we adapted ourselves to the one built for KEYBOARD by simply pretending that the current note was to be placed before the last node in the list of currently playing notes.

The INITPROG Routine

The INITPROG routine initializes the synthesizer and keyboard interface cards and sets up the CHANNEL ADDRESS array which is used to address the different channels of the synthesizer card or cards.

Addressing Synthesizer Channels

To simplify addressing of the port numbers representing the voices of the synthesizer card or cards, procedure INITPROG sets up a zero page array of numbers which represent offsets beyond the base I/O address for the Apple II ($C080). Presently there is room to set up an array for up to five synthesizer units, although if more than three are used more space for the linked list and stack used by the KEYBOARD routine must be allocated. The initial element of this array is the offset for voice 0, the second for voice 1, and so on. If more than one synthesizer unit is being used, the first voice of the second unit would be counted as voice 3 in this scheme. In other words, you may reference all available voices without regard to the host unit, since the voice number is used as an index into this array of offsets, which then gives the proper address of the voice in question. The base addresses to use to reference the proper port is simply the Apple II base I/O address plus the port number. The following are the correct base I/O addresses for the various ALF functions:

$C080: volume control

$C083: mode control A for this unit

$C084: divisor control

$C087: mode control B for this unit

Using the base address for voice 0 in conjunction with the

offset correctly accesses any port. For example, to poke a new volume in voice 2, load register Y with 2 and register A with the volume level, then execute the following code:

```
LDX    CHANADR,Y
STA    $C080,X
```

The INITSONG Routine

The INITSONG routine sets up the data structures necessary for the procedures KEYBOARD and PLAYBACK. It initializes the linked list of currently playing notes by placing the end of scan "note" in the last node of the linked list; the forward and backward pointers of this node point to itself. It also sets the expected note pointer to point to this node. Thus, until a key is pressed on the keyboard, the end of scan note will always be found in the list when the keyboard interface generates an interrupt for the end of scan.

INITSONG also sets up a stack of available nodes for the linked list. Nodes are numbered from zero to the number of voices available minus one (so that if there are three voices available, the nodes are numbered 0, 1, and 2. These numbers also represent the voice number on which a note is played. INITSONG initializes a stack pointer to point to the top of the stack, which holds the value 0.

4. HOW TO USE THE KEYBOARD PLAYING ROUTINES

A note on terminology: in the following, a CHANNEL refers to one of the three available pitch/volume outputs of a single ALF SYNTHESIZER CARD. A channel, then, represents a separate VOICE. You program a channel to output a specific pitch and volume; a voice is what you hear as a result. You program the output of a channel by storing values in specific memory locations of the Apple computer. These memory locations are called PORTS. Each synthesizer card has eight ports, numbered 0 to 7. Two ports control the mode of output (normal or pulsed); each channel on the card gets two of the remaining six ports--one to control volume, the other to control pitch. The KEYBOARD CARD is that which Doug Barnhardt built.

The keyboard card must be in slot #3. You can use as many synthesizer cards as the slots will hold; they must, however, be in consecutive slots and you have to poke the number of the leftmost slot (or the number of the slot in which a single card resides) in zero page location $A9 (decimal 169). You also have to poke the number of cards in $A8 (decimal 168).

Currently three assembly language routines are available. Each is described below.

The INITPROG routine must be called at the beginning of your program. It uses the number of synthesizer cards

and the slot number of the leftmost card (previously poked) to build a list of pointers necessary to address multi-card systems (the same list is used for a single card system, too). It also sets all available channels to normal mode and to zero volume. The number of channels (voices) available is left at $AB (decimal 171). If you want to play fewer voices than are possible (you want to hold a student down to a single note at a time, for example), poke the number you want at this location after calling INITPROG and before calling INITSONG.

The INITSONG routine must be called for each separate song played using KEYBOARD or replayed using PLAYBACK. It initializes the stack and linked list necessary to keep track of keys pressed and released.

The KEYBOARD routine gets data from the keyboard card, plays the notes via the synthesizer card, and stores music data. As it stands, KEYBOARD handles all of the music input from start to finish without interacting with its calling (presumably higher-level language) routine. It returns to its caller only when the end of song is signalled by pressing the set function button <SF> on the Paia keyboard, or the 'Q' key on the Apple console.

5. COMMUNICATION BETWEEN PASCAL AND THE LOWER LEVEL ROUTINES

Most communication of values between the Pascal host and the assembly level routines it calls takes place via the .CONST and .PUBLIC pseudo-ops of the 6502 assembler. These pseudo-ops allow, respectively, labels declared in the CONSTant and VARiable sections of the main Pascal program to be shared as sixteen bit values by the lower level assembly routines. In all cases, the assembly level routines utilize only the low-order byte of this value, so the Pascal host takes care that all values fall in the range 0 to 255. Note that the Pascal host need not treat shared labels specially in any other way--there is no equivalent directive in Pascal declaring variable labels as shareable. (This is not the case for procedure labels, which must be declared as external.)

The following labels in the main CONSTant declaration section of the Pascal host are shared by lower level routines:

1. UNITS: shared by procedure INITPROG, this is the number of synthesizer cards available.
2. SYSLOT: shared by procedure INITPROG, this is the initial slot number of the synthesizer cards.
3. KBSLOT: shared by procedure INITPROG, this is the slot number in which the keyboard interface card resides.

The following are labels declared in the VARiable section of the main Pascal host which are shared by lower level routines:

1. NUMVOC: shared by procedures INITPROG and KEYBOARD (PLAYBACK uses some of the KEYBOARD routines using this value), this is the number of voices available. INITPROG initializes this value to be the maximum number of voices possible; if fewer voices are desired, the value in this location must be changed.
2. SONG: shared by procedure INITSONG, this is a pointer variable pointing to the first byte of song data. This address value is stored in location SONGDATA by INITSONG and this location is used as an index for storing the song data by procedure KEYBOARD and for accessing the song data for replaying by procedure PLAYBACK.
3. TCLEF: shared by procedure GETBITS, this is a pointer variable which points to a packed array of boolean used as a parameter to the intrinsic routine DRAWBLOCK to draw a treble clef. Procedure GETBITS is responsible for loading the correct address in this variable.

4. BCLEF: like TCLEF, except the bits pointed to form a treble clef.

5. NBITS: shared by procedure GETBITS, this is a pointer variable which points to an array of a packed array of boolean. This array represents the bitmaps used by intrinsic procedure DRAWBLOCK to draw all musical figures except ties between notes. GETBITS is responsible for loading the correct address in this variable.

6. BEATS: shared by procedure INITSONG, this is the number of beats per measure desired by the user. INITSONG simply stores this value in the song data; it has no other purpose in the lower level routines.

7. BEATNOTE: shared by procedure INITSONG, this is the metric figure receiving a beat. Divide one by this value to obtain the appropriate figure, e.g, BEATNOTE = 1 represents a whole note, 2 a half note. INITSONG simply stores this value in the song data; it has no other purpose in the lower level routines.

8. BEATUNITS: shared by procedures INITSONG and KEYBOARD (PLAYBACK shares routines of KEYBOARD using the value), this is the number of time units (obtained by poking the game paddle) in one beat.

INITSONG simply stores this value in the song data; KEYBOARD uses it to create a metronome.

CHAPTER III

DESCRIPTION OF THE SYSTEM

The user has the ability to

1. Play music from the keyboard.
2. Save the song played on disk.
3. Load a song from disk.
4. Replay the song currently loaded.
5. Notate the song currently loaded.
6. Set musical parameters such as beat, tempo, and key.

When the program is executed, the screen is put into mixed graphics and text mode; at the top of the graphics portion is a header and the following information:

1. The title of the song currently in memory (if the song has been loaded from disk)
2. The name of the envelope currently in effect, along with the values of the envelope parameters Attack, Decay, Sustain, and Release (none of this is implemented).
3. The current values of the beat, tempo, and key. The default values are 4/4 for beat, 120 beats per minute for tempo, and C major for key.

A standard musical score with bass and treble clefs takes up the rest of the graphics screen.

The text portion of the screen is used for prompt lines, which show the user the options available at any time. At the top level of the music program there are seven options. The user sees this prompt:

```
MUSIC: R(ECORD, N(OTATE, P(LAYBACK,
       S(ET, G(ET, W(RITE, Q(UIT
```

Typing the initial letter of an option name takes the user to the corresponding "mode," which generally has a prompt of its own. Each top level mode is described below.

R(ECORD

This mode allows the user to play the Paia keyboard, generating the notes through the ALF synthesizer card. In addition, song data gets stored for later allowing the song played to be saved, notated, or played back. After typing 'R' a metronome starts clicking, if one was requested; as you press keys on the Paia keyboard, the appropriate pitch is programmed on the ALF card. The volume level is controlled by the envelope routine (for now, the pitch is heard at maximum volume at all times). At the same time, data is generated which tells which keys were pressed and for how long.

During this period of live keyboard input, the user is prompted with the line

RECORD: <SF>,Q(UIT

The user ends a keyboard input session by pressing either the <SF> button on the Paia keyboard, or the 'Q' key on the Apple board.

S(ET

This mode allows the user to alter parameters used by the keyboard playing routines. The user sees these options:

SET: B(EAT, T(EMPO, K(EY, E(NVELOPE, V(OICES, Q(UIT

1. BEAT allows the setting of a new time signature. The system responds by asking how many beats per measure are desired, then which durational unit will receive a beat. The first number should be the top number of a standard time signature, the second the bottom. For example, if 3/4 time is desired respond as follows:

   NUMBER OF BEATS PER MEASURE? 3
   FIGURE RECEIVING BEAT? 4

2. TEMPO allows the user to determine the tempo at which a song is played on the keyboard. The tempo is specified by the standard Maelzel metronome

value, i.e., the number of beats per minute. The default value is 120, or a beat every half second.

After a tempo is set, the user will be prompted by a metronome whenever he plays from the keyboard using the R(ECORD option. Note that a metronome value of 0 is legal--this means that no metronome is desired--but the N(OTATE option will not be valid if a song was played with no metronome.

3. KEY allows the user to set the key in which he wants the song to be notated. Keys are specified as 0 to 7 sharps or flats. First the user sees the following prompt line:

   KEY: S(HARP, F(LAT, Q(UIT

   Type 'S' to specify the key in sharps, 'F' for flats, or 'Q' to return with no alterations. After typing 'S' or 'F' the user will be asked for the number of accidentals desired. Any number from 0 to 7 is legal. After this, the appropriate key signature will appear on the screen.
4. ENVELOPE production is not currently implemented.
5. VOICES allows the user the set the number of voices allowed to be played simultaneously. The default number of voices the is maximum given the number of ALF synthesizer cards.

6. QUIT exits S(ET mode and takes the user back to the top command level of the Music Transcription System.

N(OTATE

This transcribes current song data into screen graphics. The song is notated a measure at a time. After a measure has been shown, the user will have the option of editing that measure (and only that measure). He sees the following prompt:

NOTATE: M(EASURE, E(DIT, Q(UIT

Typing 'M' moves the song forward one measure, scrolling the score as necessary. Typing 'E' enters edit mode for this measure. Typing 'Q' returns to the top-level command line of the music program.

If the measure is too long to show at one time on the screen, only the first part of the measure is shown at first. To see the rest of the measure, the user must hit the <return> key, as prompted by the line

<RETURN> TO CONTINUE MEASURE

Another half-screenfull of the measure will then be scrolled on the screen; in this way the previous part of the measure

is not entirely lost, allowing the user to see the relationships between it and the part currently being shown.

P(LAYBACK

This plays the current song using stored song data. If the song was input using a metronome, it is replayed with one as well. The user sees the following simple prompt during playback:

PLAYBACK: Q(UIT

which reminds him that he can stop the playback at any time by typing 'Q'.

G(ET

Reads a song from disk, placing it in memory, and making it "current" for editing, playing, or notating. The user sees the following prompt:

GET: TITLE OF SONG?

Type in the name of the song followed by <return>. The title appears at the top of the screen in the TITLE field of the header. Note, however, that the time signature and tempo of the loaded song do not change at this point, but only after entering notate mode.

W(RITE

This writes current song in memory to disk. The user sees the following prompt:

WRITE: AS WHAT SONG TITLE?

The user types in the name that he would like to give to the song and hits <return>. The song data is stored on disk under the name specified. It is given the extension .MUSIC, but this is not actually used while in the music program.

Q(UIT

This option exits the Music Transcription System.

CHAPTER IV

TRANSCRIBING LIVE KEYBOARD INPUT

1. METHOD

The process of translating live keyboard input into a graphics musical score consists of three major steps:

1. Obtaining raw song data representing keypresses and keyreleases and the real time occurring between these live events.
2. Breaking the raw song data into measures consisting of "basic events" representing note values and rests, along with information concerning their duration in musical terms rather than real time.
3. Translating the basic data from step 2 into score data representing each kind of musical structure: notes and rests having duration from whole down to one-sixty-fourth, ties, dots, accidentals, and bars.

This final product of score events tells us explicitly how the musical structure should be scored. It tells us exact x- and y-positions on the screen (and is therefore the most Apple-dependent of the steps), and it gives us indices into arrays of bitmaps representing each musical figure.

The first step is that of the keyboard preprocessor, already discussed at length. The second and third steps are discussed below.

Parsing The Song Data

Before the song can be displayed on the screen in music notation, the song data must be translated to a form more amenable to screen data, i.e., types of metric figures and x- and y-positions on the screen. For example, the raw song data tells us nothing about pauses in the music which must be notated as rests. In fact, just because we know that a key was pressed for 0.83 seconds tells us nothing about whether that note should be represented as a quarter note or as two whole notes. By the same token, we know nothing about how to break up the various musical events into measures.

The necessary translation is a two-step process. First the song data is examined and broken into "basic events"--we see that here is a note, here a rest, here a measure. We also change timing information from its raw real time form to a form more meaningful musically, namely to the duration of a basic event in terms of "timeframes" having a 1/64 note duration. The second step in the translation is changing the basic event information to "score events." This tells us that a note having the

duration specified in a basic event must be scored as, for example, an eighth note followed by a dot, and must be placed at x-position 100 and y-position 129 on the high-resolution screen. All that then needs to be done to show the measure on the screen is to pass the score event data to a routine which correctly shows the various bitmaps for each kind of musical event.

Translating Into Basic Events

Changing the raw song data as it appears from the keyboard input routine to an "intermediate code" form consisting of a linked list of basic events consists mainly of two things: deciding the duration of each basic musical event and deciding where measures break up. Of course the real time duration of each keypress can be easily obtained by simply adding the number of time units which occur between a KEYPRESS of a given key and its associated KEYRELEASE. This information alone, of course, tells us nothing about the musical structure of the musical event. However, we have required the user to specify the tempo of the music he played; this information along with the time signature information tells us everything we need to know (indeed, everything we can know) about the musical structure of the musical events represented by the song data.

The duration of a basic musical event is obtained as follows: we think of measures as being broken up into "timeframes" which have the duration of a sixty-fourth note. A measure consists of as many of these timeframes as it takes to fill the measure--in other words, a measure consists of as many timeframes as there would be sixty-fourth notes if it consisted entirely of sixty-fourth notes. For example, in 4/4 time, a measure has sixty-four timeframes; in 3/4 time, it has 48; in 4/2 time it has 128. To determine the duration of a musical event we note the timeframe (numbered 0 up to one less than the number of timeframes) in which the event starts and the timeframe in which it stops. The duration is then just the number of timeframes which elapse between the start and stop of a musical event.

The trick is, of course, using the time information gathered during real time music input to determine the timeframes in which music events occurred. We need only calculate how many real-time time units it takes to fill one timeframe. This is simply the number of time units in one measure divided by the number of timeframes in one measure. The number of time units per measure is given by

(# time units per beat) * (# beats per measure)

The number of timeframes in a measure is a function of the time signature:

((# beats per measure) * 64) / beatnote

where beatnote equals 1 if a whole note receives one beat, 2 if a half note, 4 if a quarter note, and so on.

Thus one timeframe in terms of real-time time units is given by

(# time units per measure) / (# timeframes per measure)
= ((# time units per beat) * beatnote) / 64

Naturally, we use real rather than integer arithmetic in these calculations to avoid loss of precision; and to make things easier to keep track of, we multiply through by 64, so that the number of time units in one measure is

((# time units per beat) * (# beats per measure)) * 64.

and the number of timeframes per measure is

(# time units per beat) * beatnote * 1.

The other thing that remains is deciding in which timeframe an event occurs. Assuming that we start at position zero at the beginning of each measure, we simply keep track of how many time units elapse before the event occurs. This value divided by the number of time units in a single timeframe gives us the timeframe in which the event

occurs. Now we need only calculate in which timeframe a note or rest begins and in which it ends.

Whenever a KEYPRESS command is seen, its value is inserted into a table of currently pressed keys, along with the timeframe in which it started. Then when its corresponding KEYRELEASE is seen, it is deleted from this table and output as a node in the linked list of basic events. The duration field of this node is obtained by calculating the number of timeframes which elapsed from the time the key was pressed to the time it was released.

Routine NOTATE calls routine GETMEASURE to parse a single measure upon user request. GETMEASURE examines the song data and translates it to an "intermediate code" which is a linked list of what we call basic events. A basic event simply represents what music is made of--time periods in which sounds are heard and time periods in which no sounds are heard. The former are tagged as NOTEMARKS; the latter as RESTMARKS. Each node of the list of basic events contains the following fields:

1. KIND: denotes whether the basic event is a NOTEMARK or RESTMARK. (Also STARTMARK, denoting the beginning of a song; ENDMARK, denoting the end of a song; and MEASUREMARK, denoting the end of a measure.)

2. VALUE: contains the actual adjusted keypress value (from 0 to 36) obtained from the keyboard. This field is not meaningful for kinds other than NOTEMARK.
3. DURATION: contains the length of time in timeframes of 1/64 beat that the event lasts; this field is meaningful only for notes and rests.
4. TIED: contains a pointer to another basic event node in case this node represents a carry over of an event across a measure or across a chord.
5. CHORD: contains a pointer to another basic event node which occurs during the same span of time as the current node.
6. SCRN: contains a pointer to a score event node (see below) which represents how this basic event looks graphically.
7. NEXT: contains a pointer to the following basic event node.

Translating Basic Events To Score Events

Routine GETMEASURE calls routine SHOW in turn to translate the basic event data to "score data," that is, data which is easily shown graphically. Each score event is represented as a node in another linked list which is

pointed to by the SCRN field of the basic node which this list represents. SHOW translates the duration field of the basic node into a series of notes, ties, and dots which fit the duration. It decides whether a note is represented with an upward or downward pointing flag. It keeps track of the horizontal screen positioning. It does all this to output a list of score events having the following fields:

1. KIND: this is what the whole notate routine is really all about. It represents the actual notational value, from a whole note to a sixty-fourth rest, as well as accidentals, ties, and dots. This value is used as an index into an array of bitmaps (NBITS) which represents the screen data for each score event (with the single exception of tie marks, which, since they vary greatly in length, are put on the screen using Turtlegraphics commands).
2. XPOS: the horizontal position where this score event is to be shown. The actual value may exceed the rightmost x-coordinate of the screen; routine PUTMEASURE worries about whether or not it can show the event on the physical screen. The x-position represents the left-most x-coordinate of where the bitmap of this score event must start.

3. YPOS: the vertical position where this score event is to be shown. This position is dependent on the intrinsic DRAWBLOCK routine--it gives the lower-most y-coordinate of where the bitmap of this score event must start.
4. NEXT: contains a pointer to the next node in this list of score events. A value of NIL represents the end of the list.

Screen Data Structures

The method of defining the structures used to represent the actual musical figures on the screen deserves some explanation. We use the Apple intrinsic procedure DRAWBLOCK which takes as arguments a packed Boolean array representing the figure which is to appear on the screen, the x- and y-position on the screen where the array is to be shown, and several others. The Boolean array is known as a "bitmap," since each bit of the elements of the array is used to determine whether to place a white or a black dot at a particular position on the screen.

The x- and y-positions specified represent the lower left corner of the block of bits given by the Boolean array.

The difficulty lies in defining the Boolean array. The bitmap for each musical structure is 13 bits wide by 23 bits high, which is a block of 299 bits. We certainly need a clean way to define the values of these bits for each of the durational units.

This turns out to be easy at the assembly level. This will be easier to explain by using an example. Suppose we want to define an array 16 bits wide and 3 bits high. We want the bits to look like this:

1100001111000011

0000111100001111

1111110000111111

In the first place, when defining the bitmaps we have to work bottom up; this is because DRAWBLOCK starts at the lower left corner of the block of bits and works upward. Secondly, we have to work in the usual low byte/high byte fashion. Finally, we have to work left to right when calculating the numeric value represented by the series of bits.

For example, breaking up the last line of bits for easy changing to hexadecimal digits, we have

F 3 C F

1111 1100 0011 1111

The left two represent the low byte; the right two the high byte. But the rightmost bit of each byte is the most

significant, so when calculating the numeric value of that byte, think right to left. The series of bits above then becomes

```
.BYTE 3F,FC
```

and the three lines would be defined like this:

```
.BYTE C3,C3
.BYTE F0,F0
.BYTE 3F,FC
```

All that now remains is to make the data thus defined available to the Pascal host. This is accomplished easily using Pascal pointer variables shared with the assembly language routine. By declaring the variable name PUBLIC in the assembly language routine, we can store the address of the bit definitions. We can use this label afterwards in the Pascal host as a pointer variable and pass it to DRAWBLOCK as normal.

The GETBITS Routine

The GETBITS routine is used to initialize bitmaps used to display musical symbols. It loads pointer variables shared with the Pascal host with the address of the beginning of the bit maps for the bass and treble clefs (TCLEF and BCLEF) and with the address of the beginning of an array of bitmaps for each of the other musical figures (NBITS). The rest of the routine is simply data definition using the .BYTE assembler directive.

2. SOME SPECIFIC SOLUTIONS TO TRANSCRIPTION PROBLEMS

We list below the solutions to some particularly sticky problems in the transcription of live keyboard input to a graphics musical score. All of these are problems which are outside that of deciding the musical duration of notes and rests. They include the way ties between notes are represented and scored, the way a specific sequence of musical figures is chosen to represent the duration of a note or rest, the way the pitch is used to determine placement of a musical figure on the score, and the way keys and key signatures are represented and notated.

Ties

The score event of tie is output during the third step of transcription--translating the list of basic events into a list of score events. The tie score event always follows the score event representing the right-hand part of the tie. The x- and y-position fields of the tie node have a different meaning than those of other types of score events. The x-position field is the x-position of the right hand side of the tie; the y- position field is the x-position of the left hand part of the tie. It is left to the routine actually placing the score events on the screen (PUTMEASURE) to decide the proper y-placement of the tie.

This method of placing the tie node following the right hand node of the two representing the two notes being tied is necessary to easily notate ties across bars. In this case, we do not know at the point of notating the current measure whether a note within it is tied to a note in the following measure, since we have not yet seen the next measure. Thus we wait until we see the next measure before we output the tie. In this case we set the TIED field of the basic node representing the right side of the tie to point to the node representing the left part. Then, when scoring this node we simply determine the x-position which was given to the left part and set the score node for the tie appropriately. Thus, that a note is tied across to the previous measure is flagged by its basic node having a non-nil TIED field.

The tie is the only score event which gets put on the screen using other than the DRAWBLOCK procedure. Instead, we use Turtlegraphics line drawing commands. This is necessary since a tie can have many different lengths.

Note Duration

Routine SHOW, which is responsible for translating basic events to score events, gets information about the duration of each basic event in terms of timeframes as described earlier. SHOW must translate this information

about each event into a series of nodes representing actual musical figures, such as quarter notes, ties, and dots. For example, a musical event which is a note having duration 24 timeframes must be translated into score events representing a quarter note (accounting for 16 timeframes) and a dot (representing 8 timeframes). A note having duration 20 timeframes must be represented by score events quarter note, sixteenth note, and a tie between them.

The algorithm to do this is simple. We find the largest power of two which "fits" the value of the duration. This becomes the initial score event representing the basic event. For example, the largest power of two fitting a duration of 42 is 32, which represents a one-half note (32/64). Following this initial score event, we output a series of dots and notes tied if needed; these represent the remainder of the duration. We simply take the remainder and again find the largest power of two which "fits" it. If the "fit" is exactly one-half that of the preceding score event, a dot is output. Otherwise a note followed by a tie is output. This continues until all the duration has been accounted for.

Note Placement

Each of the thirty-six keys on the Paia keyboard are assigned a unique number from 0 to 35 in ascending order from left to right. C below middle C gets 0; C# gets 1; D gets 2; and so on. Two arrays, NLINE and NBASE contain information about where each key value is to be placed on the score. NBASE represents the y-position where the initial note of an octave (beginning with C) goes; NLINE represents the offset beyond this base y-position for each key value within an octave (numbered 0 to 11). The entries of NLINE are updated according the user-requested key signature. That is, for example, the key F#/G flat can be scored either on the F or G position on the score, depending on the key signature. The appropriate information is put in the array NLINE whenever a new key is requested.

Given these two arrays, all that needs to be done is to decide which octave a key value falls in and where it falls within that octave. The y-position of the note is then calculated by adding the values in the corresponding cells of the arrays NBASE and NLINE. This information is easily obtained by integer division and remainder operations on the key value: division by 12 yields the octave number; the remainder yields the position within the octave.

Keys And Key Signatures

Keys are easily represented as sets of key values. The notes of a scale are numbered 0 to 11 (representing C to B♯). We will refer to these values as "pitch class members." The pitch class member of a key on the keyboard is easily obtained by taking the key value (0-36) modulo 12. The scoring routine does not care about keys except that they tell whether a key value must be scored with or without an accidental (and with which accidental: sharp, flat, or natural).

This information is maintained using three disjoint sets of pitch class members: a BASESET which represents those pitch class members which need no accidental (these are the values which are taken care of by the key signature); a SIGNEDSET which represents those pitch class members which must be output with an accidental corresponding to the "keymode," either sharp or flat; and a NATURALSET which represents those scale values which must be output with a natural mark. Since these three sets are disjoint, only the first two are explicitly defined--a value not included in either of these two is automatically included in the third.

Given the pitch class member, we need only decide where on the score to place the note and whether or not the note must be accompanied by an accidental. Deciding whether a note must have an accidental is trivial--we simply check

which set the pitch class member of the note occurs in. If it is a member of BASESET, the note is scored without an accidental. If it is a member of SIGNEDSET, it is scored along with the appropriate accidental, either sharp or flat. If it is in neither of these two sets, it is by default a member of NATURALSET; this means that the note is scored along with a natural mark.

The difficult part of all this is constructing the sets given a key. We use the the notion of the circle of fifths to do this. The user inputs the key in terms of a keymode (either sharp or flat) and a number, which is the number of sharps or flats occurring in the key signature. Then, starting at the right place, we simply use modular 12 arithmetic to successively add or subtract 5, depending on the keymode. The resulting number is added to BASESET and removed from SIGNEDSET (if it is there--after the fifth accidental it will not be). In addition, we remove its "corresponding" value from BASESET. Its corresponding value is determined by adding or subtracting one from its value, again depending on keymode.

For example, the key of C major corresponds to the following sets:

BASESET = [0,2,4,5,7,9,11]

SIGNEDSET = [1,3,6,8,10]

NATURALSET = []

These correspond to the white keys and black keys, with no keys needing a natural mark. Consider the key of one sharp. This means that pitch class member F# (6) is to be scored as an F with no marks on the score (its accidental occurs on the key signature); this means that pitch class member F (5) is to be scored with an accidental. Thus we add the value 6 to BASESET, remove it from SIGNEDSET, and add the value 5 to NATURALSET after removing it from BASESET. This same scheme continues for further sharps. Flats are represented in the same way.

As we alter the sets, we change the value in the arrays NBASE and NLINE (see the section on NOTE PLACEMENT above) to correspond to where the affected notes will be placed on the music score.

As a final example, the key of five sharps is represented by the following sets:

BASESET = [1,3,4,6,8,10,11]

SIGNEDSET = []

NATURALSET = [0,2,5,7,9]

CHAPTER V

RESULTS

Although it is possible to break live keyboard input into discrete musical structures, it is not possible to calculate the duration of each structure correctly in all cases. Several reasons forbid this:

1. It is impossible to tell, outside of certain limits, whether the space of time between the playing of two keys represents, for example, a short rest or just the time necessary to differentiate the notes. We have tried as far as possible to eliminate time spans of less than 1/64 note duration between two notes, adding the extra duration to one of the notes. However, it is always possible that an experienced musician actually means for there to be a very short rest between notes. Heuristics may actually add errors here.
2. Our timing mechanism is very crude, giving us errors of possibly several hundredths of a second in real time. A better timing scheme would involve a real-time hardware clock which generates interrupts at given intervals; at every clock interrupt we could poll the keyboard for keys

currently pressed. As it stands, we do exactly the opposite: the keyboard generates interrupts and we poll the clock. Given that the Apple II has no hardware clock (although one may be purchased as a peripheral), we must use the game paddle as our timing mechanism. Even so, a better timing scheme would be to scan the entire keyboard during each of our time units. This would involve processing each interrupt from the keyboard and returning immediately for the next until the end of scan is reached, at which time we would complete the time unit by successive pokes of the game paddle. We tried this and found that it simply took too much time, even in the simplest cases involving a minimal amount of processing. An entire scan usually took longer than the 1/300 second we had after an initial poke of the game paddle. The method chosen is therefore the only one feasible. Thus there will always be a certain amount of error in timing. Even when playing only one note on the keyboard, an error of one time unit is possible. This is because the key currently being processed as currently pressed have been released immediately following its interrupt. It will be at least one time unit later before we note the change. Our

heuristic involving timeframes makes up for this error as far as possible, but it has limitations, especially as the tempo is increased.

The system therefore seems best suited as an input mechanism to a musical editor. The system correctly breaks input into discrete events and places them correctly on the score; it scores the duration as well as possible. Using an editor, errors of duration can be edited and new musical events can be added if desired. An editor compatible with this system is currently under construction. It will offer not only the ability to change the score resulting from keyboard input but to input an entire piece from scratch.

BIBLIOGRAPHY

BIBLIOGRAPHY

Randolph, Nelson, "A Graphics Text Editor for Music, Part 1: Structure of the Editor," Byte, April, 1980, pp. 124-138.

_____, "A Graphics Text Editor for Music, Part 2: Algorithms," Byte, May, 1980, pp. 104-118.

Raskin, Jeff, "Using the Computer as a Musician's Amanuensis, Part 1: Fundamental Problems," Byte, April, 1980, pp. 18-28.

_____, "Using the Computer as a Musicians's Amanuensis, Part 2: Going from Keyboard to Printed Score," Byte, May, 1980, pp. 120-128.

Read, Gardner. Music Notation: A Manual of Modern Practice. Second Edition. Boston: Allyn and Bacon, Inc., 1969.

APPENDICES

APPENDIX A

ZERO PAGE USAGE: PASCAL VERSION

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$0	X	X														
$1	X															
$2																
$3																
$4																
$5																
$6																
$7																
$8																
$9																
$A								X	X	X	X					
$B	X	X	X	X	X	X	X	X	X	X						
$C	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
$D																
$E	X	X	X	X	X	X	X	X	X	X	X	X				
$F																

APPENDIX B

THE KEYBOARD PREPROCESSOR: BASIC VERSION

A version of the keyboard playing and data acquisition routines KEYBOARD, INITPROG, and INITSONG has also been developed which is compatible with Applesoft Basic. The main differences between this version and the Pascal version are these:

1. Whenever the monitor ROM is to be used under Pascal, a special "soft switch" must be accessed (which consists of a simple store operation to a specific memory location); before returning to the Pascal host, this soft switch must be flipped the other way to allow Pascal to run. None of this needs to be done under Basic.
2. Many zero page memory locations available under Pascal are not available under Basic. This called for changing many of the zero page locations to locations in the body of the code as well as switching other zero page locations.

Appendix C is a list of the memory locations relevant to the user of the Basic version of the keyboard routines.

The Basic version of the keyboard routines were written using the UCSD Pascal compatible 6502 assembler. The following is a short guide to the user who wants to change or relocate the routines. Any changes to the source text file must be made using the UCSD Pascal editor and the edited text must be assembled under the UCSD Pascal system as well. In addition, the program LOADER must be executed to load the resulting code file into memory. Here are the steps which must be taken:

1. First the ORG (orginate) directive of the KEYBOARD routine must be altered to the address where you want the code relocated. This statement is clearly marked and occurs at the beginning of the file. Assemble the resulting file; be sure to get an assembled listing because you have to note the new addresses of the following labels:
    1. INTADR
    2. EXPECTP
    3. LASTP
    4. NOTE
    5. FLINK
    6. BLINK
    7. STACK

8. STACKTOP
9. CLOCK
10. BEATS
11. BEATNOTE
12. BEATUNITS
13. MAXNOTES
14. MAXTIME

In addition you have to note the address of the last instruction of the assembled file, since routines INITPROG and INITSONG will have to be relocated as well and they currently are located directly above KEYBOARD. Some of the labels noted above are equated in routine INITSONG; these equates will have to be changed to match their new addresses. Other labels noted above are addresses which will have to be POKEd with certain values, such as the metronome value, etc.

2. Next change the ORG directive at the start of INITPROG; be sure that the address does not conflict with that of any other routines. Assemble the resulting file, again noting the address of the last statement to avoid any conflicts with any other routines, especially INITSONG which is currently located directly above INITPROG.

3. Now alter the text file of INITSONG. Change the ORG directive to the address where you want the routine to start. In addition you must change the addresses of the equates which occur at the head of the text (any other equates are in the file EQUATES.TEXT). These labels are the ones you noted in the re-assembled version of KEYBOARD. Simply change the value of the address to which each label is equated to its new value. Assemble the resulting file, again noting the address of the last instruction.
4. Now execute the file LOADER from the top-level of the Pascal system. This program will ask you for the file you want loaded--respond with the name KEYBOARD.CODE and hit <return>. Then you will be asked for the absolute address where you want the code file loaded. Respond with the decimal equivalent of the starting address you specified in the ORG directive in the KEYBOARD text file. Again hit <return>. The code file will be loaded starting at the address you specify.

   Two things may now happen. You may again be asked for the name of a file to be loaded; or you may be taken to "monitor level." Which happens depends on where you had the program loaded, since

the action of loading may destroy some memory location used by Pascal. If the former happens, just type <return> and you will be placed in the monitor.

5. At the monitor level you will save the range of memory in which the routine KEYBOARD now resides. Hook up a cassette tape recorder to the computer. Now use the tape write command to write the range of memory on tape (you know the starting and ending addresses of the range because you noted them above).
6. Now you can boot up Basic, enter the monitor level from there, read back the range of memory just saved on tape, and BSAVE the resulting memory area.
7. Now reboot Pascal and repeat steps 4 through 6 for the code files of INITPROG and INITSONG. After that you should have a binary code file in Basic of each of the three routines. You may want to BLOAD all three and BSAVE the entire memory image if the three have been located next to one another.

APPENDIX C

MEMORY ADDRESSES FOR KEYBOARD PREPROCESSOR: BASIC VERSION

These are memory locations for the version of the keyboard preprocessor dated 8/13/80:

```
SONGSTRT = $06, 6.: address of first byte of song data
SONGDATA = $08, 8.: address of last byte of song data
UNITS    = $F9, 249.: number of synthesizer units
SYSLOT   = $FA, 250.: slot in which leftmost synthesizer unit reside
BEATS    = $CF3, 3326.: number of beats per measure
BEATNOTE = $CFF, 3327.: metric value receiving one beat
BEATUNITS= $D00, 3328.: number of time units per beat
MAXNOTES = $D01, 3329.: maximum number of notes to be played
MAXTIME  = $D02, 3330.: maximum time in increments of 2.5 seconds
```

These are the current calling addresses of each routine:

```
KEYBOARD: $AA7, 2727.
INITPROG: $D70, 3440.
INITSONG: $DE0, 3552.
```

The song data can start at $E25, decimal 3621.

VITA

James M. Ross, III was born in Louisville, Kentucky on July 7, 1951. He was graduated from Oak Ridge High School in Oak Ridge, Tennessee in June 1969. He entered the University of Tennessee that year and in August 1975 received a Bachelor of Arts degree in Religious Studies.

He entered the Graduate School of The University of Tennessee, Knoxville, in September 1978. He received the Master of Science Degree with a major in Computer Science in December 1980. Mr. Ross will be employed with Bell Telephone Laboratories after graduation.