Datalink Wristapp Programmer's Reference

John A. Toebes, VIII

jtoebes@geocities.com

Table of Contents

About the DataLink

So, What is a Datalink?

The Datalink is a pretty neat watch that Timex created which allows you to download information just by pointing the watch at the display screen. You have probably seen the commercials where the dog and cat play around with reprogramming the appointments on the watch.

What makes the watch interesting to me is that you can actually write programs for it. Although Timex did not document how to do this, it turned out not to be too difficult to figure out how to write code for the watch. Of course explaining how to do that is a bit more difficult, but that is what this document is all about.

There are four basic models of the Datalink

- The original 75 model which allowed you to download phone numbers, alarms, lists, anniversaries, and appointments.
- The updated 150 model which doubled the download speed and increased the amount of memory for storing those phone numbers, alarms, lists, anniversaries, and appointments. Timex also was kind enough to give us the ability to download wristApps to extend the functionality of the watch.
- The smaller 150s model which is nearly identical to the 150 in capabilities. This was introduced for the 1996 Christmas season as a lady's watch.
- The Ironman Datalink watch. Some people called this the 150r, but that is not the correct designation. While this watch is similar to the 150 and 150s with respect to capacity, it does not support downloading of wristapps.

What programs can I load in the 150/150s?

Timex ships several useful Wristapps with the 150 in the box:

- *Note* - Used for copying up to 255 characters of text (30-40 words) to the watch. It is useful for storing directions, etc. that need to be readily accessible.
- *Melody Tester* - Used for testing Watch Tones on the watch. It sure beats waiting around for the appointment beep to go off.
- *Stopwatch* - A chronograph that times events by starting from zero and counting up.
- *Adjustable Timer* - Allows setting of a time to be counted from 1 minute to 100 hours, in 1 minute increments.
- *Preset Timer* - The Preset Countdown Timer that allows for quick selection of the following preset times: 5, 10, 15 20, 30, 45, or 60 minutes.
- *Week of the Year (U.S.)* - Displays what week of the year it is, what day of the year it is, and how many days are left in the year.
- *Week of the Year (International)* - Displays what week of the year it is, what day of the year it is, and how many days are left in the year.

You can also purchase the optional Wristapps, which give you a few other useful wristapps:

- *Golf* - A golfer's electronic scorecard. Enter the number of strokes per hole and let the watch calculate the total for the round and the front and back nine. You can recall your totals or hole scores at any time.
- *CopyMe Game* - A memory game. The watch displays a sequence of 0's that you must duplicate using the watch's buttons. If you are successful, the watch adds another step to the sequence. Make it through 15 steps and you win!
- *Pulse* - Gives you a quick estimate of your pulse rate. Feel for your pulse. When the watch beeps, start counting beats. When you count ten, press a button, and the watch calculates your pulse. It's a great workout companion.

- *World Time* - Displays the time in each of the 24 time zones around the world.
- *Conversion* - Gives you a table for converting values from one unit to another.

Datalink Technical Details

Download Protocol

Synchronization Process

Before you can start sending any data to the DataLink, you have to send a series of sync bytes:

```
$55 (the watch has to see 4 in a row to be happy about it)
```

Once the watch has gotten the Sync bytes, it will look for a series of at least four `$AA` or `$BF` bytes to go into an initialization mode.

Once in initialization mode, it will start looking for the data bytes. If it sees a `$EF`, it will treat that as an escape byte and read in the next byte regardless of what it is (this allows the first byte of the packet to be a `$55`, `$AA`, `$BF` or even `$EF`).

Once it has gone into data transfer mode, it expects a series of 2 byte groups where the low bit of the first byte and the high bit of the second byte (I call these middle bits) must match to be sync bits. It expects these sync bits to alternate between 0 and 1. Any 2-byte group that does not match this will be thrown out. Also, if no valid bytes are received within 1/5 second, the transfer operation is aborted.

Sync Bits

With these sync bits, you can only transfer 14 bits of data for every 16 bits sent. (There are actually 2 extra sync bits on the screen to act as start and stop bits). If you look at it, that means that you can get 7 bytes transferred for every 8 bytes sent. The organization of these bits is:

A	b	c	d	e	f	g	-		-	i	j	k	l	m	n	h
Q	r	s	t	u	o	p	+		+	z	y	A	B	v	w	x
G	H	I	C	D	E	F	-		-	O	P	J	K	L	M	N
W	Q	R	S	T	U	V	+		+	X	Y	Z	1	2	3	4

Where - and + represent the sync bits (zero and one) in the byte pairs. If you decode these bits into the corresponding bytes, you get:

A	b	c	d	e	f	g	h
I	j	k	l	m	n	o	p
Q	r	s	t	u	v	w	x
Y	z	A	B	C	D	E	F
G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V
W	X	Y	Z	1	2	3	4

Note that you always have to send in byte pairs, but the code is smart enough to throw away an extra byte which does not fit in a packet. All packets end with a 2 byte 16-Bit CRC.

I think that the most interesting packet of all of this is the CPACKET_JMPMEM. It is possible to reset the watch by just sending this packet in the stream:

09	23	04	3e	18	94	81	<crc-16>

What this does is tells it to jump to location 04 3e which happens to be the address of where the 4th byte in the packet is stored. The code executes the 18 94 which is a `BSET 4,TIMER_FLAGS` followed by an 81 = RTS. When the watch sees that `4,TIMER_FLAGS` has been set, it will run the watch through a complete reset cycle. There are a lot of other fun things that you can do. For example, you can play a tune during the download by storing new values at location `0335`. So the packet:

0c	23	04	3e	a6	01	c7	03	35	81	<crc-16>

Would change the download tone to be a LOW C. Replace the 01 with any value up to 0f and you can actually play a tune as it is downloading. (The note at `$0335` is played after each packet).

You can also use this code to indicate a status on the watch by setting the individual segments on the bottom:

0d	23	04	3e	a6	48	b7	1d	19	1e	81	<crc-16>

Would turn on the AM indicator. Of course since you can't look at the watch while it is downloading, it would be little silly. However, this can be a great debug aid for someone working on the download protocol since the symbols are not cleared out once the download process starts.

The CPACKET_MEM packet is also pretty useful. You can use it to set any of the locations in ram to a particular value. This might be useful if you know that you have a certain Wristapp already loaded and you want to change some data stored in the wristapp. All you need is the address to store the data in and the data that you want to put there.

Packet Format

$20 - CPACKET_START

0	Packet Length
1	$20 - CPACKET_START
2	$00
3	$00
4	Version: 3=V2.0 for the 150, 4=V2.1 for the 150s
5	CRC-16 High
6	CRC-16 Low

$21 - CPACKET_SKIP

This skip packet does get sent to the Datalink, but its contents are completely ignored.

0	Packet Length
1	$21 – CPACKET_SKIP
2	<ignored>
3	CRC-16 High
4	CRC-16 Low

$23 - CPACKET_JMPMEM

This JMPMEM packet is useful for jumping to/calling specific locations in memory during the download process.

0	Packet Length
1	$23 – CPACKET_JMPMEM
2	Address High to jump to
3	Address low to jump to
4	CRC-16 High

5	CRC-16 Low

$90 - CPACKET_SECT

This is the Initialization packet to start loading a section. There are three formats based on the section to be loaded.

$90 - CPACKET_SECT - Format 1

0	Packet Length
1	$90 – CPACKET_SECT
2	$01 - CLOAD_EEPROM - Load up EEProm data
3	Number of CPACKET_DATA packets to follow
4	CRC-16 High
5	CRC-16 Low

$90 - CPACKET_SECT - Format 2

0	Packet Length
1	$90 – CPACKET_SECT
2	$02 – CLOAD_WRISTAPP - Load a new Wristapp
3	Number of CPACKET_DATA packets to follow
4	Value to be stored in COMM_010e
5	CRC-16 High
6	CRC-16 Low

$90 - CPACKET_SECT - Format 3

0	Packet Length
1	$90 – CPACKET_SECT
2	$03 - CLOAD_SOUND - Load a new sound scheme
3	Number of CPACKET_DATA packets to follow
4	Base offset for the sound (should be $100-length of the sound)
5	CRC-16 High
6	CRC-16 Low

$91 - CPACKET_DATA

This is the data packet sent after a CPACKET_SECT. The number of packets sent will be dependent on the section and is indicated in the CPACKET_SECT packet. Once these packets start getting sent, there should be no other packets until a CPACKET_END is encountered (although there is really no error checking done on it). If the download is terminated without the last CPACKET_END being seen or the right number of CPACKET_DATA packets, the entire section is ignored.

0	Packet Length
1	$91 - CPACKET_DATA
2	<ignored> (probably address high)
3	<ignored> (probably address low)
4 .. n+4	n Databytes to be stored
n+5	CRC-16 High

n+6	CRC-16 Low

$92 - CPACKET_END

This packet marks the end of a section.

0	Packet Length
1	$92 - CPACKET_END
2	Section (1=CLOAD_EEPROM, 2=CLOAD_WRISTAPP, 3=CLOAD_SOUND)
3	CRC-16 High
4	CRC-16 Low

$93 - CPACKET_CLEAR

This Packet is used to clear out a section.

0	Packet Length
1	$93 - CPACKET_CLEAR
2	Section to clear (CLOAD_EEPROM, CLOAD_WRISTAPP, CLOAD_SOUND)
3	CRC-16 High
4	CRC-16 Low

$50 - CPACKET_ALARM

This packet is used to set the alarm information for a single alarm.

0	Packet Length
1	$50 - CPACKET_ALARM
2	Alarm Number (1-5)
3	Alarm Hour (0-23)
4	Alarm Minute (0-59)
5	<ignored>
6	<ignored>
7	Alarm String character 1
8	Alarm String character 2
9	Alarm String character 3
10	Alarm String character 4
11	Alarm String character 5
12	Alarm String character 6
13	Alarm String character 7
14	Alarm String character 8
15	Alarm enable 0=disable, non-zero=enable
16	CRC-16 High
17	CRC-16 Low

$32 - CPACKET_TIME

This single packet is used to set the time. It should be sent early in the process in ensure the best synchronization with the CPU clock time.

0	Packet Length

1	$32 – CPACKET_TIME
2	Time zone selector (1=Time zone 1)
3	Seconds (0-59)
4	Hour (0-23)
5	Minute (0-59)
6	Month of the year (1-12)
7	Day of the month (1-31)
8	Current year (mod 1900)
9	Time Zone Name character 1
10	Time Zone Name character 2
11	Time Zone Name character 3
12	Day of the week (0=Monday...6=Sunday)
13	12/24 hour selector (1=12 Hour format, anything other than 1=24 hour format)
14	Time zone date format
15	CRC-16 High
16	CRC-16 Low

$70 - CPACKET_MEM

This packet is used to store a number of bytes into memory at a fixed location. Note that it is not used for loading up a wristapp because other information has to be reset when a wristapp has been loaded.

0	Packet Length
1	$70 – CPACKET_MEM
2	High byte of memory address
3	Low byte of memory address
4..n+3	Data to be stored into memory
n+4	CRC-16 High
n+5	CRC-16 Low

$71 - CPACKET_BEEPS

This packet is used to control the hourly chimes and button beep flags.

0	Packet Length
1	$71 - CPACKET_BEEPS
2	Enable Hourly chimes flag (0=Disable, Non-Zero=Enable)
3	Enable Button beep flag (0=Disable, Non-Zero=Enable)
4	CRC-16 High
5	CRC-16 Low

The Display

The DataLink display has 4 basic areas when it comes to programming. For convenience, I call them simply:

1. *TOP* - The top 6 digits. Each of these digits are represented by 9 segments which can be individually controlled. There are dash and period separators between the second/third and the forth/fifth digits for displaying dates. There is also a dash separator between the third and forth digits which is used for telephone numbers. It also has a tic mark before the first digit as a shorthand for the first two digits of the year.

2. *SYMBOLS* - The AM/PM, Reminder, Night Mode, Alarm, and Note symbols. These tend to only be used by the Time app.

3. *MIDDLE* - Like the TOP area, the Middle area also consists of 6 digits each made up of 9 segments. For separators between the second and third digits, you can use a colon, period, or a dash. The fourth and fifth digits can be separated by a dash or a period.

4. *BOTTOM* - The bottom 8 digits which are each represented by a 5 by 5 matrix of pixels that can be individually addressed. The ROMs also support a series of scrolling routines to allow a message to be scrolled across the bottom at a nice even rate.

What is really nice about the watch is that every segment on the display is individually addressable. For convenience, we use a notation of BIT:OFFSET to indicate how to address the segment. What this means is that you need to set DISP_ROW (`$001d`) to the OFFSET value and then set/clear the BIT in DISP_COL (`$001e`) to turn on/off the corresponding segment. For example, if you wanted to turn on the AM indicator on the 150 which is referred to as `4:48`, you would do:

```
LDA   #$48
STA    DISP_ROW
BSET   4,DISP_COL
```

Here's a quick overview of the display. All of the segments are clickable so that you can determine the way to set/clear that segment. This is a Java applet, so if your browser is not capable of supporting Java, you won't be able to see it. When you click on the segment, it will hi-light in red and display the appropriate set values on the status bar. Value1 will be what you use for the 150 and Value2 will be for the 150S.

The TOP/MIDDLE Character Set

The TOP and MIDDLE lines only allow for 32 different characters to be displayed (unless of course you do it all yourself). For convenience, we refer to this character set as the TIMEX6 character set. All of the Wristapps that are written use the TIMEX6 macro to convert ASCII strings to this set. Because you have to use the number zero for the letter O and the number five for the letter S, the TIMEX6 macro will handle the conversion for you. The characters that can't easily be displayed are: J K Q V X Y. Fortunately, they aren't used in a lot of words (except of course my first name :-).

The TIMEX6 character set does allow for the names of all the internal Apps to be displayed. It is important to be aware of this limited character set when choosing the name of your Wristapp, otherwise you won't be able to display it easily when someone switches to the app.

$00	$01	$02	$03	$04	$05	$06	$07	$08	$09	$0a	$0b	$0c	$0d	$0e	$0f
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

$10	$11	$12	$13	$14	$15	$16	$17	$18	$19	$1a	$1b	$1c	$1d	$1e	$1f
G	H	:	L	M	N	P	R	T	U	W	Y	r	_	-	+

The routines which are useful for putting strings on the top and middle lines are:

PUT6TOP	???
PUT6MID	???
PUTMSG1	???
PUTMSG2	???
CLEARTOP	???
CLEARMID	???
CLEARTOP12	???
CLEARTOP34	???
CLEARTOP56	???
CLEARMID12	???
CLEARMID34	???
CLEARMID56	???
PUTLINE1	???
PUTLINE2	???
PUTTOP12	???
PUTTOP34	???
PUTTOP56	???
PUTMID12	???
PUTMID34	???
PUTMID56	???

You can see what all of these are displayed as below.

<table>
<tr><th>$00 – 0</th><th>$01 - 1</th><th>$02 - 2</th><th>$03 - 3</th></tr>
<tr><td>|=====|
| |
| |
| |
| |
| |
|=====|</td><td> |
      |
      |
      |
      |
      |
      |</td><td> =====|
      |
      |
 =====|
|
|
|=====</td><td> =====|
      |
      |
 =====|
      |
      |
 =====|</td></tr>
</table>

<table>
<tr><th>$04 - 4</th><th>$05 - 5</th><th>$06 - 6</th><th>$07 - 7</th></tr>
<tr><td>| |
| |
| |
|=====|
      |
      |
      |</td><td>|=====
|
|
|=====
      |
      |
 =====|</td><td>|=====
|
|
|=====
| |
| |
|=====|</td><td> =====|
      |
      |
      |
      |
      |
      |</td></tr>
</table>

<table>
<tr><th>$08 - 8</th><th>$09 - 9</th><th>$0a - A</th><th>$0b - B</th></tr>
<tr><td>|=====|
| |
| |
|=====|
| |
| |</td><td>|=====|
| |
| |
|=====|
      |
      |</td><td>|=====|
| |
| |
|=====|
| |
| |</td><td> =====|
    | |
    | |
 =====|
    | |
    | |</td></tr>
</table>

<table>
<tr><td>|=====|</td><td>=====|</td><td>| |</td><td>=====|</td></tr>
</table>

<table>
<tr><th>$0c - C</th><th>$0d - D</th><th>$0e - E</th><th>$0f - F</th></tr>
<tr><td>|=====
|
|
|
|
|
|=====</td><td>=====|
  | |
  | |
     |
  | |
  | |
=====|</td><td>|=====
|
|
|=====
|
|
|=====</td><td>|=====
|
|
|=====
|
|
|</td></tr>
</table>

<table>
<tr><th>$10 - G</th><th>$11 - H</th><th>$12 - I</th><th>$13 - L</th></tr>
<tr><td>|=====
|
|
|
| |
| |
|=====|</td><td>| |
| |
| |
|=====|
| |
| |
| |</td><td>
   |
   |

   |
   |
</td><td>|
|
|
|
|
|
|=====</td></tr>
</table>

<table>
<tr><th>$14 - M</th><th>$15 - N</th><th>$16 - P</th><th>$17 - R</th></tr>
<tr><td>|=====|
| | |
| | |
| |
| |
| |
| |</td><td>|=====|
| |
| |
| |
| |
| |
| |</td><td>|=====|
| |
| |
|=====|
|
|
|</td><td>|=====|
| |
| |
|=====|
| |
| |
|</td></tr>
</table>

<table>
<tr><th>$18 - T</th><th>$19 - U</th><th>$1a - W</th><th>$1b - Y</th></tr>
<tr><td> =====
   |
   |

   |
   |
</td><td>| |
| |
| |
| |
| |
| |
|=====|</td><td>| |
| |
| |
| |
=====|</td><td>|
=====| | | </td></tr>
</table>

<table>
<tr><th>$1c - r</th><th>$1d -</th><th>$1e - -</th><th>$1f - +</th></tr>
<tr><td>

 =====
|
|
|</td><td></td><td>

=====

</td><td>
   |
   |
=====
   |
   |
</td></tr>
</table>

The Bottom Character set

The BOTTOM line has a slightly richer character set which we call the TIMEX character set. It allows for 64 different characters, includes the entire upper case alphabet and quite a few special symbols. All of these characters are drawn on a 5x5 dot matrix.

$00	$01	$02	$03	$04	$05	$06	$07	$08	$09	$0a	$0b	$0c	$0d	$0e	$0f
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

$10	$11	$12	$13	$14	$15	$16	$17	$18	$19	$1a	$1b	$1c	$1d	$1e	$1f
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V

$20	$21	$22	$23	$24	$25	$26	$27	$28	$29	$2a	$2b	$2c	$2d	$2e	$2f
W	X	Y	Z	_	!	“	#	>	%	&	‘	(	)	*	+

$30	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3a	$3b	$3c	$3d	$3e	$3f
,	-	.	/	:	\	DIV	=	BELL	?	_	CHK	PREV	NEXT	BLOCK	SEP

The routines which are useful for putting strings on the top and middle lines are:

BANNER8	???
PUTMSGXBOT	???
PUTMSGBOT	???
PUTBOT678	???
PUTLINE3	???
PUT_LETTERX	???
PUTSCROLLMSG	???
SCROLLMSG	???
SCROLLMSG_CONT	???

$00-0	$01-1	$02-2	$03-3	$04-4	$05-5	$06-6	$07-7
__@@_ _@__@ _@__@ _@__@ __@@_	__@__ _@@__ __@__ __@__ __@__	@@@@_ ____@ @@@ @____ @@@@@	@@@@_ ____@ @@@ ____@ @@@@_	@__@_ @__@_ @@@@@ ___@_ ___@_	@@@@@ @____ @@@@_ ____@ @@@@_	_@@@@ @____ @@@@_ @___@ @@@	@@@@@ ____@ ___@_ __@__ __@__

$08-8	$09-9	$0a-A	$0b-B	$0c-C	$0d-D	$0e-E	$0f-F
_@@@_ @___@ _@@@_ @___@ _@@@_	_@@@_ @___@ _@@@@ ____@ @@@@_	_@@@_ @___@ @@@@@ @___@ @___@	@@@@_ @___@ @@@@_ @___@ @@@@_	_@@@@ @____ @____ @____ _@@@@	@@@@_ @___@ @___@ @___@ @@@@_	@@@@@ @____ @@@@_ @____ @@@@@	@@@@@ @____ @@@@_ @____ @____

$10-G	$11-H	$12-I	$13-J	$14-K	$15-L	$16-M	$17-N
_@@@@ @____ @_@@@ @___@ _@@@@	@___@ @___@ @@@@@ @___@ @___@	@@@ __@__ __@__ __@__ @@@	__@@@ ___@_ ___@_ @__@_ _@@__	@___@ @__@_ @_@__ @@_@_ @___@	@____ @____ @____ @____ @@@@@	@___@ @@_@@ @_@_@ @_@_@ @___@	@___@ @@__@ @_@_@ @__@@ @___@

$18-O	$19-P	$1a-Q	$1b-R	$1c-S	$1d-T	$1e-U	$1f-V
@@@ @___@ @___@ @___@ @@@	@@@@_ @___@ @@@@_ @____ @____	@@@ @___@ @_@_@ @__@_ _@@_@	@@@@_ @___@ @@@@_ @__@_ @___@	_@@@@ @____ @@@ ____@ @@@@_	@@@@@ __@__ __@__ __@__ __@__	@___@ @___@ @___@ @___@ @@@	@___@ @___@ @___@ _@_@_ __@__

$20-W	$21-X	$22-Y	$23-Z	$24-	$25-!	$26-“	$27-#
@___@ @___@ @_@_@ @@@ _@_@_	@___@ _@_@_ __@__ _@_@_ @___@	@___@ _@_@_ __@__ __@__ __@__	@@@@@ ___@_ __@__ _@___ @@@@@	_____ _____ _____ _____ _____	__@__ __@__ __@__ _____ __@__	_@_@_ _@_@_ _____ _____ _____	_@_@_ @@@@@ _@_@_ @@@@@ _@_@_

$28-$	$29-%	$2a-&	$2b-‘	$2c-(	$2d-)	$2e-*	$2f-+
_@@@@ @_@__ @@@ __@_@ @@@@_	@@__@ ___@_ __@__ _@___ @__@@	_@___ @_@__ _@@_@ @__@_ _@@_@	__@__ _@___ _____ _____ _____	__@__ _@___ _@___ _@___ __@__	__@__ ___@_ ___@_ ___@_ __@__	@_@_@ @@@ @@@@@ @@@ @_@_@	__@__ __@__ @@@@@ __@__ __@__

$30-,	$31--	$32-.	$33-/	$34-:	$35-\	$36-	$37-=
_____ _____ _____ __@__ _@___	_____ _____ @@@@@ _____ _____	_____ _____ _____ _____ __@__	____@ ___@_ __@__ _@___ @____	_____ __@__ _____ __@__ _____	@____ _@___ __@__ ___@_ ____@	__@__ _____ @@@@@ _____ __@__	_____ @@@@@ _____ @@@@@ _____

$38->	$39-?	$3a-_	$3b-	$3c-	$3d-	$3e-	$3f-
__@__ _@@@_ _@@@_ @@@@@ __@__	_@@__ @__@_ __@__ _____ __@__	_____ _____ _____ _____ @@@@@	____@ ___@_ @_@__ _@___ @@@@@	____@ __@@@ @@@@@ __@@@ ____@	@____ @@@__ @@@@@ @@@__ @____	@@@@@ @@@@@ @@@@@ @@@@@ @@@@@	_____ _@@@_ _@@@_ _@@@_ _____

Alarm	4:1C	4:1A
AM	4:48	4:46
M1A	4:42	4:40
M1B	3:40	3:3E
M1C	2:40	2:3E
M1D	2:46	2:44
M1E	3:46	3:44
M1F	4:46	4:44
M1G	3:44	3:42
M1H	4:44	4:42
M1I	2:44	2:42
M2A	4:3A	4:38
M2B	3:38	3:36
M2C	2:38	2:36
M2D	2:3E	2:3C
M2E	3:3E	3:3C
M2F	4:3E	4:3C
M2G	3:3C	3:3A
M2H	4:3C	4:3A
M2I	2:3C	2:3A
M3A	4:30	4:2E
M3B	3:2E	3:2C
M3C	2:2E	2:2C
M3D	2:34	2:32
M3E	3:34	3:32
M3F	4:34	4:32
M3G	3:32	3:30
M3H	4:32	4:30

M3I	2:32	2:30
M4A	4:28	4:26
M4B	3:26	3:24
M4C	2:26	2:24
M4D	2:2C	2:2A
M4E	3:2C	3:2A
M4F	4:2C	4:2A
M4G	3:2A	3:28
M4H	4:2A	4:28
M4I	2:2A	2:28
M5A	4:1E	4:1C
M5B	3:1C	3:1A
M5C	2:1C	2:1A
M5D	2:22	2:20
M5E	3:22	3:20
M5F	4:22	4:20
M5G	3:20	3:1E
M5H	4:20	4:1E
M5I	2:20	2:1E
M6A	4:10	4:0E
M6B	3:0E	3:0C
M6C	2:0E	2:0C
M6D	2:14	2:12
M6E	3:14	3:12
M6F	4:14	4:12
M6G	3:12	3:10
M6H	4:12	4:10
M6I	2:12	2:10
MC23	3:36	3:34
MD45	3:24	3:22
MP23	2:36	2:34
MP45	2:24	2:22
Night	4:26	4:24
Note	4:0e	4:0C
PM	4:40	4:3E
Remind	4:38	4:36
S1A1	2:47	
S1A2	2:45	
S1A3	2:43	
S1A4	2:41	
S1A5	2:3F	
S1B1	3:47	
S1B2	3:45	
S1B3	3:43	
S1B4	3:41	

S1B5	3:3F	
S1C1	4:47	
S1C2	4:45	
S1C3	4:43	
S1C4	4:41	
S1C5	4:3F	
S1D1	0:47	
S1D2	0:45	
S1D3	0:43	
S1D4	0:41	
S1D5	0:3F	
S1E1	1:47	
S1E2	1:45	
S1E3	1:43	
S1E4	1:41	
S1E5	1:3F	
S2A1	2:3D	
S2A2	2:3B	
S2A3	2:39	
S2A4	2:37	
S2A5	2:35	
S2B1	3:3D	
S2B2	3:3B	
S2B3	3:39	
S2B4	3:37	
S2B5	3:35	
S2C1	4:3D	
S2C2	4:3B	
S2C3	4:39	
S2C4	4:37	
S2C5	4:35	
S2D1	0:3D	
S2D2	0:3B	
S2D3	0:39	
S2D4	0:37	
S2D5	0:35	
S2E1	1:3D	
S2E2	1:3B	
S2E3	1:39	
S2E4	1:37	
S2E5	1:35	
S3A1	2:33	
S3A2	2:31	
S3A3	2:2F	
S3A4	2:2D	

S3A5	2:2B	
S3B1	3:33	
S3B2	3:31	
S3B3	3:2F	
S3B4	3:2D	
S3B5	3:2B	
S3C1	4:33	
S3C2	4:31	
S3C3	4:2F	
S3C4	4:2D	
S3C5	4:2B	
S3D1	0:33	
S3D2	0:31	
S3D3	0:2F	
S3D4	0:2D	
S3D5	0:2B	
S3E1	1:33	
S3E2	1:31	
S3E3	1:2F	
S3E4	1:2D	
S3E5	1:2B	
S4A1	2:27	
S4A2	2:25	
S4A3	2:23	
S4A4	2:21	
S4A5	2:1F	
S4B1	3:27	
S4B2	3:25	
S4B3	3:23	
S4B4	3:21	
S4B5	3:1F	
S4C1	4:27	
S4C2	4:25	
S4C3	4:23	
S4C4	4:21	
S4C5	4:1F	
S4D1	0:27	
S4D2	0:25	
S4D3	0:23	
S4D4	0:21	
S4D5	0:1F	
S4E1	1:27	
S4E2	1:25	
S4E3	1:23	
S4E4	1:21	

S4E5	1:1F	
S5A1	2:1D	
S5A2	2:1B	
S5A3	2:19	
S5A4	2:17	
S5A5	2:15	
S5B1	3:1D	
S5B2	3:1B	
S5B3	3:19	
S5B4	3:17	
S5B5	3:15	
S5C1	4:1D	
S5C2	4:1B	
S5C3	4:19	
S5C4	4:17	
S5C5	4:15	
S5D1	0:1D	
S5D2	0:1B	
S5D3	0:19	
S5D4	0:17	
S5D5	0:15	
S5E1	1:1D	
S5E2	1:1B	
S5E3	1:19	
S5E4	1:17	
S5E5	1:15	
S6A1	2:13	
S6A2	2:11	
S6A3	2:0F	
S6A4	2:0D	
S6A5	2:0B	
S6B1	3:13	
S6B2	3:11	
S6B3	3:0F	
S6B4	3:0D	
S6B5	3:0B	
S6C1	4:13	
S6C2	4:11	
S6C3	4:0F	
S6C4	4:0D	
S6C5	4:0B	
S6D1	0:13	
S6D2	0:11	
S6D3	0:0F	
S6D4	0:0D	

S6D5	0:0B	
S6E1	1:13	
S6E2	1:11	
S6E3	1:0F	
S6E4	1:0D	
S6E5	1:0B	
S7A1	2:09	
S7A2	2:07	
S7A3	2:05	
S7A4	2:03	
S7A5	2:01	
S7B1	3:09	
S7B2	3:07	
S7B3	3:05	
S7B4	3:03	
S7B5	3:01	
S7C1	4:09	
S7C2	4:07	
S7C3	4:05	
S7C4	4:03	
S7C5	4:01	
S7D1	0:09	
S7D2	0:07	
S7D3	0:05	
S7D4	0:03	
S7D5	0:01	
S7E1	1:09	
S7E2	1:07	
S7E3	1:05	
S7E4	1:03	
S7E5	1:01	
S8A1	2:02	
S8A2	2:04	
S8A3	2:06	
S8A4	2:08	
S8A5	2:0a	
S8B1	3:02	
S8B2	3:04	
S8B3	3:06	
S8B4	3:08	
S8B5	3:0a	
S8C1	4:02	
S8C2	4:04	
S8C3	4:06	
S8C4	4:08	

S8C5	4:0a	
S8D1	1:02	
S8D2	1:04	
S8D3	1:06	
S8D4	1:08	
S8D5	1:0a	
S8E1	0:02	
S8E2	0:04	
S8E3	0:06	
S8E4	0:08	
S8E5	0:0a	
T1A	2:42	2:40
T1B	1:40	1:3E
T1C	0:40	0:3E
T1D	0:42	0:40
T1E	0:46	0:44
T1F	1:46	1:44
T1G	1:42	1:40
T1H	1:44	1:42
T1I	0:44	0:42
T2A	2:3A	2:38
T2B	1:38	1:36
T2C	0:38	0:36
T2D	0:3A	0:38
T2E	0:3E	0:3C
T2F	1:3E	1:3C
T2G	1:3A	1:38
T2H	1:3C	1:3A
T2I	0:3C	0:3A
T3A	2:30	2:2E
T3B	1:2E	1:2C
T3C	0:2E	0:2C
T3D	0:30	0:2E
T3E	0:34	0:32
T3F	1:34	1:32
T3G	1:30	1:2E
T3H	1:32	1:30
T3I	0:32	0:30
T4A	2:28	2:26
T4B	1:26	1:24
T4C	0:26	0:24
T4D	0:28	0:26
T4E	0:2C	0:2A
T4F	1:2C	1:2A
T4G	1:28	1:26

T4H	1:2A	1:28
T4I	0:2A	0:28
T5A	2:1E	2:1C
T5B	1:1c	1:1A
T5C	0:1c	0:1A
T5D	0:1e	0:1C
T5E	0:22	0:20
T5F	1:22	1:20
T5G	1:1E	1:1C
T5H	1:20	1:1E
T5I	0:20	0:1E
T6A	2:10	2:0E
T6B	1:0e	1:0C
T6C	0:0e	0:0C
T6D	0:10	0:0E
T6E	0:14	0:12
T6F	1:14	1:12
T6G	1:10	1:0E
T6H	1:12	1:10
T6I	0:12	0:10
TD23	1:36	1:34
TD34	4:2E	4:2C
TD45	1:24	1:22
TP23	0:36	0:34
TP45	0:24	0:22

Memory Map

The Datalink is controlled by a custom 6805 which has 16K of ROM, 1.25K of Ram and 2.0K of EEProm. Because the 6805 has a 15 bit address bus, all accesses wrap at 0800 to 0000 and repeat once again. The EEProm is a serial device and does not appear in the accessible address space for the 6805.

Datalink Overview Memory Map

`0000-002A`	6805 Hardware registers
`002B-004F`	Unused ram (probably not even mapped)
`0050-005F`	System App local variables
`0060-0067`	Wristapp local variables
`0068-00C2`	System local variables
`00C3-00FF`	Call stack
`0100-010F`	EEProm control variables
`0110-0335`	Wristapp memory
`0336-0435`	Sound memory (starts high, low end can be used for a larger wristapp)
`0436-04FF`	System upper ram
`0500-3FFF`	Unused - This is a hole in the address space
`0400-7FFF`	System ROM

<More memory map stuff to come>

Differences between the 150 and 150S

For Christmas 1996, Timex introduced a smaller version of the 150 called the 150s. This watch has substantially the same hardware and capabilities as the 150, but in a smaller package. You can tell the difference between the two by entering COMM Mode. If the version on the bottom line is V2.0, then the watch is a 150. If it says V2.1, it is a 150S. There is also a newer release of the Datalink software (V2.1) for the 150s which works with all of the Datalink watches. The older V2.0 software will not talk to the 150s.

It is not possible to run the same wristapp on both watches because of a few differences:

- The addresses for the display segments have changed. Mostly this has been a simple subtraction of 2 from the offsets for addressing the display segments, but it also involved the shuffling of a couple of the pixels in the segments on the bottom line. Since turning on a segment is a hard coded constant, an application has to be recompiled for the different display.

- To accommodate the change in display segments, a couple of ROM routines have been changed. This resulted in a shuffling of the addresses of a number of routines within the watch.

- To further complicate things, the order of a few routines in the ROM has been changed. While the routines are exactly the same in both the 150 and the 150s, the location of these routines is never the same.

- The CPACKET_START packet has a 4 for the version code instead of a 3.

- Even with all this shuffling, the memory map for the low ram appears to be exactly identical, as does the actual 6805 hardware.

Dealing with the Differences

Because the two watches are so different, you have to essentially write the same program twice with different targets for all of the system routines and any segment poking that is done. The V2.1 software handles this by storing both copies of the code in the .ZAP file with a description field to identify which watch the software is targeted to. When you identify the type of the watch to the DataLink software, it automatically chooses the right software to send.

Accessing the EEPROM

The 2K EEProm in the Datalink is accessed over a serial interface and is not directly mapped to the 6805 address space. The entries for all of the apps are stored sequentially in the EEProm with a length/flag byte at the front of each one. When an entry is deleted, it is done by simply setting the high bit on the flag byte. All of the internal software simply skips over the entry. There is no code in the watch for shuffling the data in the EEProm.

When any data is downloaded to the EEProm, it essentially clears the EEProm pointers and starts again. This has the effect of deleting all Phone, List, Anniversary, and Appointment entries if you just load a single entry down to the watch. However, the actual data in the EEProm is never cleared out except when new data overwrites. This means that it is possible to dump out the data in the EEProm even if the watch has been reset or only one or two entries downloaded to it.

There are some internal routines for getting to the EEProm (to be documented later) and it is possible with some work to write code that allows you to store entries in the EEProm, but you would have to figure out how to shuffle the entries in the EEProm if you wanted to add an entry without deleting everything (this isn't really as difficult as it sounds).

Sound Hardware

The Datalink is capable of playing 14 tones by poking one of the following values into PORT_SOUND (location $0028). From experimentation, it appears that only the low nibble of whatever value is poked into this location is actually used.

It is my current working theory that there is a timer routine in the Datalink which is actually causing the resulting frequencies and it might be possible to generate other sounds by going through a slightly different mechanism to poke the sound hardware.

Note that if you use the built-in sound routines for playing sounds, you will find that the interrupt routines will happily readjust the hardware tones behind your back.

Hardware Tones

0	Tone_END - This seems to generate silence
1	Low C
2	High C
3	Middle C
4	Very High C
5	High F (Reported to be a little bit lower than F)
6	Middle F
7	Low F
8	Very High G# (G-Sharp)
9	High G# (G-Sharp)
10	Middle G# (G-Sharp)
11	Low G# (G-Sharp)
12	High D
13	Middle D
14	Low D
15	Silence

Important Terms:

Sound Scheme - A set of sounds (this is the .SPC file in the SND directory of the Datalink application) which are downloaded to the watch. A sound scheme contains all the Soundlets and Sound Sequences for all 10 defined system sound values. This file is loaded in the watch so that the end of it is at $0435 in memory.

System Sound - is one of the 10 defined system sound values:

Value	Symbol	Purpose
$80	SND_NONE	No sound at all
$c1	SND_BUTTON	Button Beep
$c2	SND_RETURN	Return to time
$83	SND_HOURLY	Hourly Chime
$c4	SND_CONF	Confirmation
$85	SND_APPT	Appointment Beep
$86	SND_ALARM	Alarm Beep

$87	SND_DLOAD	Program Download
$88	SND_EXTRA	Extra sound
$89	SND_COMERR	Comm Error
$8a	SND_DONE	Comm done

Sound Sequence - The sequence of soundlets which are played to for a given System Sound. There can be as few as 1 Sound Sequence and as many as 10 different Sound Sequences. Each System Sound maps to one Sound Sequence although the same Sound Sequence can be used for more than one System Sound. A Sound Sequence is represented by two series of numbers.

The first series is called the Soundlet Count Table and consists of a series of one or more bytes where the last byte in the series has the high bit set ($80). For each entry in the Soundlet Count Table, the number of times that a sound is played is determined by clearing the tip bit and then using the resulting number as a count. So $81 indicates the last entry with a repeat count of 1. $A0 indicates the last entry with a repeat count of 20. $0A indicates an entry (with at least one more following it) with a repeat count of 10.

The second series is the Soundlet Pointer Table which consists of exactly the same number of entries as the Soundlet Count Table. Each entry in this table is simply a pointer to the start of the corresponding Soundlet

Soundlet - A sequence of Notes terminated by a 0 note. There is no practical limit on the number of notes in a Soundlet except for the total size of 256 bytes for the entire Sound Scheme.

Note - A single sound to be played. The note consists of a single byte broken into two Nibbles. The high order nibble is the tone to be played and the low order nibble is the duration for that tone in 1/10th of a second intervals.

Tone - One of 14 tones supported by the sound hardware on the watch as well as the two values which produce silence:

Sound Scheme Format

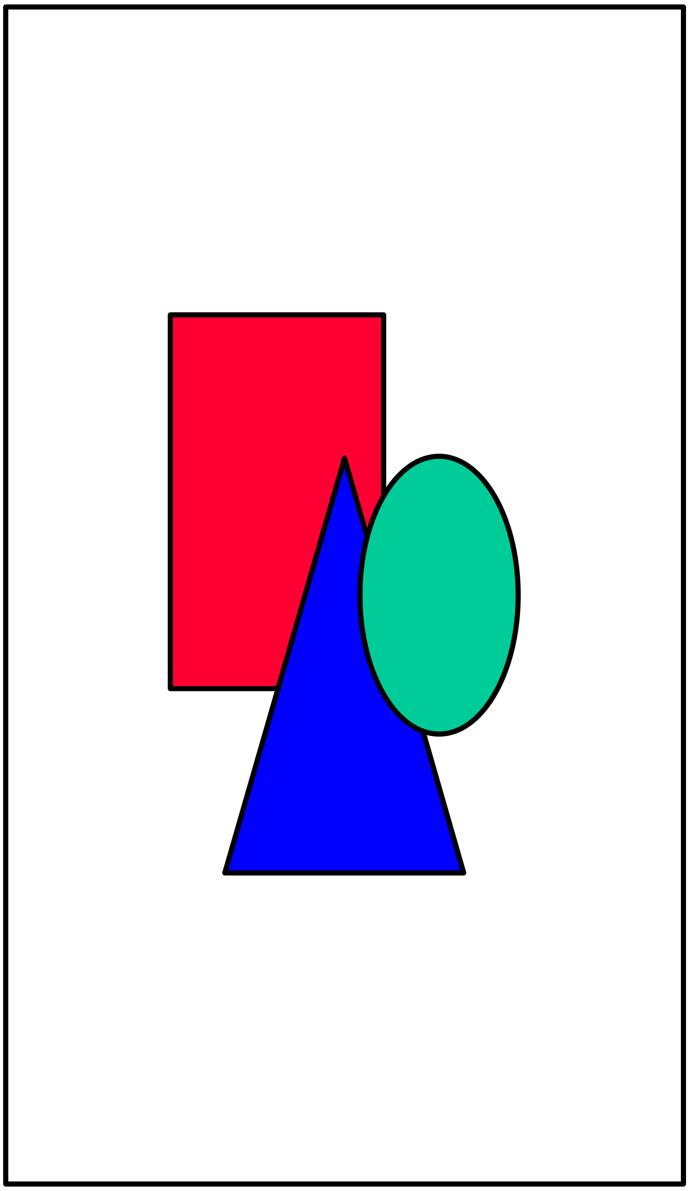
Given the default sounds in the ROM, I propose that this is how we would interpret and code them:

```
TONE_END        EQU     $00     ; END
TONE_LOW_C      EQU     $10     ; Low C
TONE_HI_C       EQU     $20     ; High C
TONE_MID_C      EQU     $30     ; Middle C
TONE_VHI_C      EQU     $40     ; Very high C
TONE_HI_F       EQU     $50     ; High F (little bit lower than F)
TONE_MID_F      EQU     $60     ; Middle F
TONE_LO_F       EQU     $70     ; Low F
TONE_VHI_GSHARP EQU     $80     ; Very High G# (G Sharp)
TONE_HI_GSHARP  EQU     $90     ; High G#
TONE_MID_GSHARP EQU     $A0     ; Middle G#
TONE_LO_GSHARP  EQU     $B0     ; Low G#
TONE_HI_D       EQU     $C0     ; High D
TONE_MID_D      EQU     $D0     ; Middle D
TONE_LO_D       EQU     $E0     ; Low D
TONE_PAUSE      EQU     $F0     ; Pause
;
; This is the default sound table
;
DEF_SOUNDS
        db      SP_1-SD_1       ; 0000: 08

        db      SD_1-DEF_SOUNDS ; 0001: 0b   BUTTON BEEP
        db      SD_2-DEF_SOUNDS ; 0002: 0c   RETURN TO TIME
        db      SD_3-DEF_SOUNDS ; 0003: 0d   HOURLY CHIME
        db      SD_4-DEF_SOUNDS ; 0004: 0e   CONFIRMATION
        db      SD_5-DEF_SOUNDS ; 0005: 0f   APPOINTMENT BEEP
        db      SD_5-DEF_SOUNDS ; 0006: 0f   ALARM BEEP
        db      SD_5-DEF_SOUNDS ; 0007: 0f   PROGRAM DOWNLOAD
        db      SD_5-DEF_SOUNDS ; 0008: 0f   EXTRA
        db      SD_6-DEF_SOUNDS ; 0009: 11   COMM ERROR
        db      SD_7-DEF_SOUNDS ; 000a: 12   COMM DONE
;
; This is the soundlet count table which contains the duration
; counts for the individual soundlets
;
SD_1    db      SND_END+1       ; 000b: 81
SD_2    db      SND_END+1       ; 000c: 81
SD_3    db      SND_END+2       ; 000d: 82
SD_4    db      SND_END+4       ; 000e: 84
SD_5    db      10,SND_END+24   ; 000f: 0a a8
SD_6    db      SND_END+10      ; 0011: 8a
SD_7    db      SND_END+16      ; 0012: a0
;
; This is the soundlet pointer table which contains the pointers to the soundlets
;
SP_1    db      SL_2-DEF_SOUNDS ; 0013: 1d
```

```
SP_2    db      SL_1-DEF_SOUNDS ; 0014: 1b
SP_3    db      SL_3-DEF_SOUNDS ; 0015: 1f
SP_4    db      SL_2-DEF_SOUNDS ; 0016: 1d
SP_5    db      SL_4-DEF_SOUNDS ; 0017: 22
        db      SL_5-DEF_SOUNDS ; 0018: 27
SP_6    db      SL_5-DEF_SOUNDS ; 0019: 2a
SP_7    db      SL_2-DEF_SOUNDS ; 001a: 1d
;
; These are the soundlets themselves.  The +1 or other number
indicates the duration for the sound.
;
SL_1    db      TONE_HI_GSHARP+1                ; 001b: 91
        db      TONE_END                        ; 001c: 00

SL_2    db      TONE_MID_C+1                    ; 001d: 31
        db      TONE_END                        ; 001e: 00

SL_3    db      TONE_MID_C+2                    ; 001f: 32
        db      TONE_PAUSE+2                    ; 0020: f2
        db      TONE_END                        ; 0021: 00

SL_4    db      TONE_HI_C+2                     ; 0022: 22
        db      TONE_PAUSE+2                    ; 0023: f2
        db      TONE_HI_C+2                     ; 0024: 22
        db      TONE_PAUSE+10                   ; 0025: fa
        db      TONE_END                        ; 0026: 00

SL_5    db      TONE_HI_C+2                     ; 0027: 22
        db      TONE_PAUSE+2                    ; 0028: f2
        db      TONE_END                        ; 0029: 00

SL_6    db      TONE_HI_C+3                     ; 002a: 23
        db      TONE_MID_C+3                    ; 002b: 33
        db      TONE_END                        ; 002c: 00
;
; This is the tone that the comm app plays for each record
;
db      TONE_MIDC/16                    ; 002d: 03
```

Sound Files

The sound scheme stored in a file is nearly identical with the exception of a 4 byte header. Given the default sound, you might picture it as below (with thanks to Pigeon for his first representation of this).



Brent Davidson gives a pretty good explanation of this: ("Absolute offset" refer to the offset location in the file. "Relative offset" refers to the location without the "header" (25 04 19 69).

The 08 at absolute offset 0004 indicates that the soundlet count table is 8 bytes long. In this case, we have only 7 different sounds, but one sound has two entries because it uses two soundlets.

The next 10 bytes represent the relative offsets of the sound sequences. The relative offset of each byte reflects the system sound it represents. This table is fixed in size because there are only 10 system sounds.

The next 8 (or however many are indicated by absolute offset 0004) bytes (the soundlet count table) are in the relative offsets pointed to by the sound sequence table. The high order nibble of the byte indicates the last entry for this sound. If it is clear, there are more soundlets associated with this sound. The remaining 7 bits in the byte are the number of times that the corresponding soundlet is to be played. Hence, a value of 0a indicates that the corresponding soundlet is to be played 10 times and the next entry in the soundlet count table is to be used for the sound. A value of 81 indicates that the corresponding soundlet is to be played once.

The next 8 bytes (or however many are indicated by absolute offset 0004) are the soundlet pointer table. They are parallel to the previous 8 bytes, and reference the relative offsets of the soundlets.

The remainder of the bytes (except for the final byte) are the soundlets themselves. The high order nibble indicates the tone, the low order nibble indicates the duration. A byte of 00 signals the end of each soundlet.

The low order nibble of the final byte of the file indicates the tone played after each record is downloaded during transmission, it's high order nibble is always 0, and it's count cannot be set.

Wristapp Programming Reference

The Processor

The Datalink contains a custom Motorola 6805 processor which performs all of the watch functions. This turns out to be a very convenient thing as the 6805 is well documented and actually pretty fun to program (IMHO). If you are looking for technical information, I tend to look to Motorola's 6805 home page and to the instruction set card Oxford University Computing Laboratory's Microprocessor reference card. All of my work has been done with just these two information sources.

To summarize the 6805, it has two 8-bit registers (A and X) and a small number of addressing modes. Since it has a 15 bit address bus, you are left with the interesting problem of using a register as a pointer. To deal with this, you have to resort to self modifying code. If you are only having to point to a small amount of memory, you can also use the indexed mode where the register is an offset from some base location. Of course, if you only have to point to things in the first 256 bytes of ram, you can pretend that a register might be a pointer.

	Bit Manip		Branch	Read/Modify/Write					Control		Register/Memory					
	BTB	BSC	REL	DIR	INH	INH	INH	IX	INH	INH	IMM	DIR	EXT	IX2	IX1	IX
	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	BRSET0	BSET0	BRA	NEG	NEGA	NEGX	NEGX	NEG	RTI		SUB	SUB	SUB	SUB	SUB	SUB
x1	BRCLR0	BCLR0	BRN						RTS		CMP	CMP	CMP	CMP	CMP	CMP
x2	BRSET1	BSET1	BHI								SBC	SBC	SBC	SBC	SBC	SBC
x3	BRCLR1	BCLR1	BLS	COM	COMA	COMX	COMX	COM	SWI		CPX	CPX	CPX	CPX	CPX	CPX
x4	BRSET2	BSET2	BCC	LSR	LSRA	LSRX	LSRX	LSR			AND	AND	AND	AND	AND	AND
x5	BRCLR2	BCLR2	BCS								BIT	BIT	BIT	BIT	BIT	BIT
x6	BRSET3	BSET3	BNE	ROR	RORA	RORX	RORX	ROR			LDA	LDA	LDA	LDA	LDA	LDA
x7	BRCLR3	BCLR3	BEQ	ASR	ASRA	ASRX	ASRX	ASR		TAX		STA	STA	STA	STA	STA
x8	BRSET4	BSET4	BHCC	LSL	LSLA	LSLX	LSLX	LSL		CLC	EOR	EOR	EOR	EOR	EOR	EOR
x9	BRCLR4	BCLR4	BHCS	ROL	ROLA	ROLX	ROLX	ROL		SEC	ADC	ADC	ADC	ADC	ADC	ADC
xA	BRSET5	BSET5	BPL	DEC	DECA	DECX	DECX	DEC		CLI	ORA	ORA	ORA	ORA	ORA	ORA
xB	BRCLR5	BCLR5	BMI							SEI	ADD	ADD	ADD	ADD	ADD	ADD
xC	BRSET6	BSET6	BMC	INC	INCA	INCX	INCX	INC		RSP		JMP	JMP	JMP	JMP	JMP
xD	BRCLR6	BCLR6	BMS	TST	TSTA	TSTX	TSTX	TST		NOP	BSR*	JSR	JSR	JSR	JSR	JSR
xE	BRSET7	BSET7	BIL						STOP		LDX	LDX	LDX	LDX	LDX	LDX
xF	BRCLR7	BCLR7	BIH	CLR	CLRA	CLRX	CLRX	CLR	WAIT	TXA		STX	STX	STX	STX	STX

```
* BSR Is a REL type instruction

INH - Inherent (1 Byte)
IMM - Immediate (2 Bytes)  e.g. LDA #20
DIR - Direct (2 Bytes)  e.g. LDA $61
EXT - Extended (3 Bytes) e.g. LDA $0244
REL - Relative (2 Bytes) e.g. BEQ *+20
BSC - Bit Set/Clear (2 bytes)e.g. BSET2 $61
BTB - Bit test and Branch (3 bytes) e.g. BRCLR2 $61,*+10
IX - Indexed (1 byte) e.g. ADD ,X or ADD 0,X
IX1 - Indexed 1 byte offset (2 bytes)  e.g. LDA $61,X
IX2 - Indexed 2 byte offset (3 bytes) e.g. LDA $0122,X
```

Tools

Unfortunately, there really aren't a lot of tools out there for creating wristapps... While there are free assemblers available on Motorola's 6805 home page, you will find that the lack of support for Timex's character set can be a bit limiting. Even more problematic is that you have to figure out how to get the program to the watch in order to run it.

My solution has been to write my own assembler which creates the .zap file format that is understood by the Datalink software on the PC. This DLZap program is pretty braindead in many ways and has quite a few bugs associated with refreshing the screen. It also is limited to creating apps only for the 150 or the 150s one at a time. If you want to create an app which runs on both watches, you have to combine them by hand.

I am working on a newer tool which doesn't have the refresh bugs (yeah, right :-) and automatically creates both the 150 and the 150s applications. Hopefully, this should be available in a couple of weeks. (Like I ever got a chance to actually finish it , but read on :-).

ASM6805 (2 months later)

Instead of fixing the refresh problems in DLZap, I realized that I needed something to address all of the work I was having to do to create wristApps and make it a bit easier (and hopefully more reproducible). I have gone and created a new version of the DLZap program which takes .zsm files and outputs the proper .zap file. Basically in a nutshell what it does is:

1. Compile from a single .zsm file and create both the 150 and the 150S versions of a Wristapp
2. Find the location where the Datalink software is installed and put the new wristapp there
3. Automatically update the timexdl.dat file to incorporate the wristapp
4. Integrate into Microsoft Developer studio to allow you to advance through errors with the F4 key.
5. Run as a windows app and allow you to select the file to assemble from a file requester

You can download the setup program for the beta here.

The .ZSM file Format

So, what is the .ZSM file format? It is nothing more than a standard .ASM file with a couple of comment lines at the beginning. For example the header for TIPCALC would be:

```
;Name: Tim Calculator
;Version: TIPCALC1
;Description: The tip calculator - by John A. Toebes, VIII
;
;Press the set button to enter the amount.  When in set mode, press the MODE button to switch between dollars and cents mode.
;Press the set button to go back to the display mode. The tip amount will scroll across the bottom of the screen as 15%, 20% and then 10% in sequence.
;
;When in display mode, pressing the prev or next buttons will enter the set mode automatically on the dollars amount.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
```

The keywords are immediately after the semicolon and before the colon. The only recognized keywords are HEADER, NAME, VERSION, DESCRIPTION, HELPFILE, HELPTOPIC, and PARENT. It uses the VERSION keyword to identify the name of the created wristapp. The remaining information is just copied into the .ZAP file for use by the Timex software. In the process of doing this, I discovered that the last digit of the first line of a .ZAP file (the line that looks like TDL0405971) indicates whether the app is a 150-only app (last digit =1) or a 150/150S Dual app (last digit=2).

System Routine Definitions

To compile a Wristapp, you need a header file which defines all of the routines that you might call. For now, I have two versions of the same file - Wristapp.i which I put into two separate directories:

- Wristapp.i for the 150
- Wristapp.i for the 150s

Program Layout Basics

Wristapp Interface Entries

Unlike more complex operating systems and modern programming environments, the Datalink Wristapps are simply a series of bytes to be loaded into the watch. They are always loaded at $0110 and there is no relocation whatsoever. This means that if you want to have more than one Wristapp in the watch at a time, you can't. However, you can get around this limitation by creating a Wristapp which performs more than one function. The biggest issue with this will be the limited amount of ram ($0110 up $0436 minus however much you use for a sound scheme). This works out to 804 bytes if you could have no sound scheme loaded. Since the typical sound scheme is about 32 bytes, a more reasonable limit is 770 bytes for a wristapp - not a lot of room for sloppy code.

0110	WRIST_MAIN	This is a JMP instruction to your primary initialization entry point for the wristapp. It is called immediately after the wristapp has been loaded for the first time and never again.
0113	WRIST_SUSPEND	This is a JMP instruction to your suspend entry point. It is called if your app is suspended because an alarm has gone off or your app has timed out because nothing has happened for 3 minutes. If you don't care about this, the three bytes should be a RTS followed by two NOP instructions.
0116	WRIST_DOTIC	This is a JMP instruction to your callback handling routine. It is called in any situation where the app has requested a callback for timed events such as the normal TIC (1/10th second), Second change, Minute change, Hour Change, and Day change. If you do not want to handle these events, the three bytes should be a RTS followed by two NOP instructions
0119	WRIST_INCOMM	This is a JMP instruction to your COMM suspend routine. It is called when the COMM app wants to suspend your Wristapp which has requested a callback for timed events. This gives your app a chance to forget about timers for a while. Note that it is possible that the app may never be reentered if the user downloads a new wristapp on top of it. If you don't care about this, the three bytes should be a RTS followed by two NOP instructions.
011C	WRIST_NEWDATA	This is a JMP instruction to your new data handling routine. It is called when the COMM app has downloaded new data to the watch. This can be useful if you have an app that has to know about the data in the EEProm such as a password protect utility. If you don't care about this, the three bytes should be a RTS followed by two NOP instructions.
011F	WRIST_GETSTATE	This is always two instructions: `LDA STATETAB,X` RTS Which are used to get an entry from **The State Table**. The X register points to the entry that is to be retrieved. You MUST supply this routine in order for the Wristapp to even function.
0123	WRIST_JMP_STATE0	This is a JMP to the state 0 handling routine.
0126	WRIST_OFF_STATE0	This is the offset into the state table for the state data associated with state 0. Unless you reorder the states, this will always be 0.
0127	WRIST_JMP_STATE1	This is a JMP to the state 1 handling routine (if any).
012A	WRIST_OFF_STATE1	This is the offset into the state table for the state data associated with state 1 (if any)

This sequence of JMP instructions followed by the offset value repeats for all of the states that your Wristapp supports. If you only have a single state, then your code can start at 0127.

Strings and Data

With any typical program, you want to be able to write to the display. If you can get away with using strings from the ROM, then you don't have to worry about where to put the strings. However, when you want to put your own strings there, you need to be aware that the BANNER8, PUT6TOP, and PUT6MID routines all take offsets from 0110 as the string to put on the display. This effectively limits you to putting all of your strings at the start of the Wristapp. Since you also know that you can't put a string until 0127, those first bytes of addressability are lost, limiting you to a total of 233 bytes of strings that you can store.

.ZAP File Format

The Timex Datalink software on the PC stores all of the Wristapps in a .ZAP file. The format of this file turns out to be pretty simple. In fact, you can edit it using any standard text editor as long as you remember that the last line can not have a Carriage return after it. This seems to make the Datalink software not always recognize the file.

Within the file, each section is terminated by a ¬ character ($AC). You can optionally put a comment on the line immediately after the separator character. For the V2.1 software, the .ZAP file contains the code for both the 150 and the 150s. For the earlier 2.0 software, the 150 code happens to be first and the 150s code is simply ignored. This allows the same .zap file to work for both versions of the software.

Applet file header	This is some sort of a version string associated with the creation time. It is typically of the form "TDLmmddyyn" where mmddyy is the date that the applet was created and n is a sequence number. The actual value of this string seems to be ignored.
Name 150	This is the name of the applet as it is to appear in the Wristapps list for the 150. The name can be any number of characters (there may be an upper limit on it) and can contain spaces and other special characters.
Version 150	This is the version number of the 150 applet. It should be up to 8 characters of alphanumeric characters. It is not clear that this is actually used by the software.
Description 150	This is the description for the 150 applet that is shown when you select it in the Wristapp panel. The description can be pretty much any length and even include blank lines. The software does its best to wrap this description when it displays it.
Help Filename 150	This is the name of the Windows .hlp file that is to be used when the user asks for help on the 150 applet. The default file that timex uses for all of its wristapps is WATCHAPP.HLP. You should provide a .hlp file for any wristapp which tells the user how the Wristapp works on the watch.
Help Index 150	This is the index in the help file associated with the help for the 150 applet. This is passed along with the Help Filename to the Windows Help system.
Config App 150	This is the configuration program (if any) that is to be invoked when the user selects the configure button in the Wristapps software. This program should be a standalone Windows program that modifies the applet as appropriate. If the program is not configurable, the string should be "none"
Watch 150	This is the name of the watch that this applet is targeted at. It should be "Timex Data Link 150 Watch"
Code 150	This is the hex code for the 150 applet. It is simply the ASCII dump of the hex digits (0-9A-Z) of the code to be downloaded to the watch. It really should be a single line of text with no spaces, but it does appear to allow the line to wrap. Since the longest this line can ever be is 1608 characters, there really isn't any need to wrap the line.
CRC 150	This is the CRC-16 associated with the 150 applet. It is only a CRC on the Code 150 string.
Data Indicator	This is the indicator of data for the 150 applet. If there is no data, this should be a 0, otherwise it is a 1.

150	
Data 150	(OPTIONAL) This is the data for the 150 applet. This entry is present ONLY if the Data Indicator 150 value is 1.
Name 150s	This is the name of the applet as it is to appear in the Wristapps list for the 150s. The name can be any number of characters (there may be an upper limit on it) and can contain spaces and other special characters.
Version 150s	This is the version number of the 150s applet. It should be up to 8 characters of alphanumeric characters. It is not clear that this is actually used by the software.
Description 150s	This is the description for the 150s applet that is shown when you select it in the Wristapp panel. The description can be pretty much any length and even include blank lines. The software does its best to wrap this description when it displays it.
Help Filename 150s	This is the name of the Windows .hlp file that is to be used when the user asks for help on the 150s applet. The default file that Timex uses for all of its wristapps is WATCHAPP.HLP. You should provide a .hlp file for any wristapp which tells the user how the Wristapp works on the watch.
Help Index 150s	This is the index in the help file associated with the help for the 150 applet. This is passed along with the Help Filename to the Windows Help system.
Config App 150s	This is the configuration program (if any) that is to be invoked when the user selects the configure button in the Wristapps software. This program should be a standalone Windows program which modifies the applet as appropriate. If the program is not configurable, the string should be "none"
Watch 150s	This is the name of the watch that this applet is targeted at. It should be "Timex Data Link 150s Watch"
Code 150s	This is the hex code for the 150s applet. It is simply the ASCII dump of the hex digits (0-9A-Z) of the code to be downloaded to the watch. It really should be a single line of text with no spaces, but it does appear to allow the line to wrap. Since the longest this line can ever be is 1608 characters, there really isn't any need to wrap the line.
CRC 150s	This is the CRC-16 associated with the 150s applet. It is only a crc on the Code 150s string.
Data Indicator 150s	This is the indicator of data for the 150s applet. If there is no data, this should be a 0, otherwise it is a 1.
Data 150s	(OPTIONAL) This is the data for the 150s applet. This entry is present ONLY if the Data Indicator 150 value is 1.

Getting Started

When your program is first invoked, you have to set a bit to tell the Roms that you are ready to handle processing. To do this, you need to set bit 7 in the WRISTAPP_FLAGS ($96). At this time, you probably want to set a few of the other requests to indicate how your Wristapp wants to process things. The bits in this flag byte are interpreted as:

WRISTAPP_FLAGS - $96

7	Wristapp has been loaded	SET=LOADED
6	Uses system rules for button beep decisions	SET=System Rules
5	Play button beep sound on wristapp for mode button	SET=ENABLE
4	Play button beep sound on wristapp for any button	SET=ENABLE
3	wristapp wants a call once a day when it changes (WRIST_DOTIC)	SET=CALL
2	wristapp wants a call once an hour when it changes (WRIST_DOTIC)	SET=CALL
1	wristapp wants a call once a minute when it changes (WRIST_DOTIC)	SET=CALL
0	wristapp wants a second timer function called at start of interrupt (WRIST_DOTIC)	SET=CALL

The State Table

An app is generally run through events passed in to it. These events are controlled by a series of state tables which indicate which events are to put the app into what state and how long to process that app for. A state table consists of a single byte followed by a series of three byte entries with a EVENT_END terminator byte after the last entry. Each entry has three parts to it:

1. The event code which indicates what event is to be accepted by this state table

2. The timer indicator to indicate how long to wait before firing off a timer if no other event occurs before it. The values can be found in the table below

3. The new state to enter when this event is encountered

The initial byte is the state to enter if an event is encountered which does not match any entry in the table.

Special State Tables

State table 0 is always entered first for an app. It will almost always have an EVT_ENTER entry in it so that you can know when an application is first called.

If an app supports nesting (all WristApps might), then it will be entered by a call to State Table 1 with an EVT_NEST event. All other state tables are completely defined by the application and may be used in any way that you want. Often a separate state is used for each mode that the app might have (such as a set mode). In order to switch between states, either you code the new state with the event, such as with the EVT_SET operation OR you can post a user event which has an associated entry in the state table that has the new state for that event.

There are two special state values associated with an event. $FF is used to indicate that the app wishes to exit and go to the next app. For WristApps, this means go back to the time app. $FE is a special value used to handle returning from a EVT_NEST nesting. If all of the nested app processing occurs in state 1, then this value would appear for an entry in the state1 table. For all others, it is assumed to be the new state table to select. No error checking is done on any of these values.

One very nice thing that can be done with the events is posting a timer to go off if no other event occurs after the current event. There are two timers although only one can be active at a time. The reason for this is to allow the app to quickly distinguish between which event timed out without having to save some global variable. These timer values are fixed in the ROM and you select which timer interval you want through the value you set. For a strange happenstance, all of the intervals of the second timer are also available for the first time (but I would be careful not to count on that).

Nested Apps

One important event that an application should handle is the EVT_RESUME which occurs after a nested app terminates. This allows your application to pick up after an alarm or appointment has gone off. When you get this event, it is a pretty good idea to refresh the display since you don't know what state the other app left it in. You should also use this time to restore any system flags that you may have set. You should also be aware that before your app is suspended, the system will call your suspend function at WRIST_SUSPEND ($0113). That will be your chance to save any variables that you expect to have trashed.

Button Events

For the events, there are three forms of the button events. The EVT_NEXT, EVT_MODE, EVT_SET, EVT_PREV, and EVT_GLOW events allow you to see when the corresponding button is pressed. When you get one of these events, you will not get notification of when the button was released. There is a set of events EVT_DNNEXT, EVT_DNMODE, EVT_DNSET, EVT_DNPREV and EVT_DNGLOW which give you the down transition for those buttons and the corresponding set of events EVT_UPNEXT, EVT_UPMODE, EVT_UPSET (I like that name), EVT_UPPREV, and EVT_UPGLOW which tell you when the button has been released. It is the case that the UP event can be handled by a different state than the DN event.

If you want to get any of those buttons, you can look for EVT_ANY (and EVT_DNANY, EVT_UPANY) which will call when any of the 5 buttons have been pressed. In order to figure out which button was pressed, your code will need to look at BTN_PRESSED ($04c3) which will contain one of the EVT_NEXT, EVT_MODE, EVT_SET, EVT_PREV, and EVT_GLOW values. Often an application does not have an interest in the Indiglo button but cares about the other 4 buttons. For this, you can use EVT_ANY4 (and EVT_DNANY4, EVT_UPANY4) just the same way as the EVT_ANY events.

Timer Events

The EVT_TIMER1 and EVT_TIMER2 events come in when the timer associated with a particular event has elapsed without another event being posted. There is no requirement of using a particular timer for a given event other than to allow you to distinguish between which event occurred. The two timers have slightly different values for when they go off and that might slightly affect your choice of timers (but that is rare). From experimentation, it appears that the time cycle for the TIMER1 is a bit slower than that for Timer2. I recommend that you use Timer2 for any of the fast actions and timer1 for the slower ones (like timing out the display).

Other Events

The EVT_USER0, EVT_USER1, EVT_USER2, and EVT_USER3 events are for an application to use for anything it wants to. Most of the time, these are useful for transitioning to a different state. You can post an event by calling POSTEVENT.

The only other event is EVT_IDLE. This event is sent only to the TIME app when another app has been suspended because it was idle for more than three minutes. Since a wristapp could never get this event, it is probably worth ignoring.

Event Constants

Here are the constants which you would find useful in creating your app:

State Table Values

EVT_NEXT	$00	Next button pressed (not interested in the up transition)
EVT_MODE	$01	Mode button pressed (not interested in the up transition)
EVT_SET	$02	Set/Delete button pressed (not interested in the up transition)
EVT_PREV	$03	Prev button pressed (not interested in the up transition)
EVT_GLOW	$04	Indiglo button pressed (not interested in the up transition)
EVT_ANY	$05	Any button pressed (not interested in the up transition)
EVT_ANY4	$06	Any button pressed except Indiglo (not interested in the up transition)
EVT_IDLE	$19	This is only sent to the TIME app when another app has been idle for more

		than three minutes
EVT_RESUME	$1a	Called when resuming from a nested app
EVT_ENTER	$1b	Initial state.
EVT_NEST	$1c	The state table 1 entry called when a nested application is called. It is the equivalent of EVT_ENTER for an interrupt. This only occurs for WristApps, Timer, and appt apps.
EVT_END	$1d	End of event table indicator
EVT_TIMER1	$1e	Timer event - This is fired for the TIM1_ values
EVT_TIMER2	$1f	Timer event - This is fired for the TIM2_ values
	$20-$36	UNUSED (I bet that you can have user specified events for these too)
EVT_USER0	$37	User specified events. Queued by calling POSTEVENT
EVT_USER1	$38	User specified events. Queued by calling POSTEVENT
EVT_USER2	$39	User specified events. Queued by calling POSTEVENT
EVT_USER3	$3a	User specified events. Queued by calling POSTEVENT
	$3b-$7f	UNUSED
EVT_DNNEXT	$80	Next button pressed
EVT_DNMODE	$81	Mode button pressed
EVT_DNSET	$82	Set/Delete button pressed
EVT_DNPREV	$83	Prev button pressed
EVT_DNGLOW	$84	Indiglo button pressed
EVT_DNANY	$85	Any of the four buttons Pressed
EVT_DNANY4	$86	Any button pressed except Indiglo
	$87-$9F	UNUSED
EVT_UPNEXT	$A0	Next button released
EVT_UPMODE	$A1	Mode button released
EVT_UPSET	$A2	Set/Delete button released
EVT_UPPREV	$A3	Prev button released
EVT_UPGLOW	$A4	Indiglo button released
EVT_UPANY	$A5	Any of the four buttons Released
EVT_UPANY4	$A6	Any button Released except Indiglo

Timer Constants

TIM_ONCE	$ff	No time interval. Operation is executed just once
TIM1_TIC	$00	
TIM1_2TIC	$01	
TIM1_3TIC	$02	
TIM1_4TIC	$03	
TIM1_HALFSEC	$04	
TIM1_SECOND	$05	
TIM1_SECHALF	$06	
TIM1_TWOSEC	$07	
TIM1_TWOSEC1	$08	
TIM1_12SEC	$09	
TIM1_18SEC	$0a	
TIM2_TIC	$80	This is the typical scroll interval

TIM2_2TIC	$81	
TIM2_4TIC	$82	
TIM2_8TIC	$83	This is the normal blink interval
TIM2_12TIC	$84	Just over a second
TIM2_16TIC	$85	A second and a half
TIM2_24TIC	$86	Two and a half seconds
TIM2_32TIC	$87	Just over three seconds
TIM2_40TIC	$88	Four seconds
TIM2_48TIC	$89	Almost five seconds
TIM2_96TIC	$8a	Almost ten seconds

Note that the second part of this table is happen-stance since it is really a rollover of the second table on top of the first one. But it might be useful to someone...

TIM1_TICA	$0b	This is the typical scroll interval
TIM1_2TICA	$0c	
TIM1_4TICA	$0d	
TIM1_8TIC	$0e	This is the normal blink interval
TIM1_12TIC	$0f	Just over a second
TIM1_16TIC	$10	A second and a half
TIM1_24TIC	$11	Two and a half seconds
TIM1_32TIC	$12	Just over three seconds
TIM1_40TIC	$13	Four seconds
TIM1_48TIC	$14	Almost five seconds
TIM1_96TIC	$15	Almost ten seconds

Classes of Callable Functions

I have broken down the system routines into 14 basic categories. For each function listed, you will find the name of the routine followed by two hex addresses separated by a slash. The first address is the location of the routine for the Datalink 150 and the second is the location for that routine on the 150s.

Anniversary support	General routines for accessing the Anniversary data in the EEProms and setting all of the flags and display to indicate the anniversaries.
Appointment support	General routines for accessing the appointment data in the EEProms and setting all of the flags and display segments for appointments.
Blinking routines	????
Event support	????
Format Routines	Routines for converting numbers into the corresponding display digits.
Indiglo support	Routines for turning on and off the Indiglo light as well as managing the timers for the light
INST Support	????
Internal	Not quite sure why you would ever call these routines, but the MIGHT be useful sometimes.
Line routines	????
Packet/EEProm Support	????
Scanning support	????
Scrolling Messages	????
Sound Support	????
Update functions	????

Anniversary support routines

Routine	**FIND_ANNIV_TODAY** - $40CD/$40BC
Parameters	None
Purpose	This finds the next anniversary entry which is greater than or equal to today

Routine	**FIND_ANNIV_SCAN** - $40D3/$40C2
Parameters	ANNIVSCAN_MONTH, ANNIVSCAN_YEAR, ANNIVSCAN_DAY - Date to scan for anniversary entry
Purpose	This finds the next anniversary entry which is greater than or equal to the scan date

Routine	**ANNIV_NEXT_ENTRY** - $40E1/$40D0
Parameters	ANNIV_CURRENT – The current anniversary entry
Purpose	Advance to the next anniversary entry. If we hit the end of the list, we need to wrap the year and go to the next one

Routine	**ANNIV_PREV_ENTRY** - $4117/$4106
Parameters	ANNIV_CURRENT – The current anniversary entry

Purpose	Advance to the previous anniversary entry. If we hit the end of the list, we need to wrap the year and go to the end again

Routine	**`FIND_ANNIV_ENTRY`** `- $415F/$414E`
Parameters	ANNIVTEST_MONTH, ANNIVTEST_DAY, ANNIVTEST_YEAR - Date of anniversary to find
Purpose	This finds the next anniversary entry which is greater than or equal to the specified date

Routine	**`CHECK_ANNIVERSARIES`** `- $41FC/$41EB`
Parameters	None
Purpose	This code checks all anniversaries to see if any occur today

Routine	**`SET_ANNIVTEST_TODAY`** `- $423A/$4229`
Parameters	None
Purpose	Latches the current month, date, year into the ANNIVTEST_ locations

Routine	**`INIT_ANNIVERSARY_DATA`** `- $4282/$4271`
Parameters	None
Purpose	This clears the ANNIVERSARY occurrence flags and latches in the current date for the anniversary check routine

Routine	**`TEST_ANNIVERSARY`** `- $4288/$4277`
Parameters	EXTRACTBUF – Anniversary data to be checked ANNIVTEST_MONTH, ANNIVTEST_DAY, ANNIVTEST_YEAR - Current date to check against
Purpose	This tests the anniversary against the current day and sets the 4,ANNIV_FLAGS and 5,ANNIV_FLAGS flags appropriately.

Routine	**`ANNIV_COPY_INFO`** `- $4308/$42F7`
Parameters	ANNIV_YEAR - The year to fake the appointment as
Purpose	This copies the current appointment information into the ANNIVSCAN variables

Routine	**`READ_ANNIV_CURRENT`** `- $4317/$4306`
Parameters	ANNIV_CURRENT – the anniversary entry to be read
Purpose	This reads in the current anniversary entry into EXTRACTBUF

Routine	**`READ_ANNIV_FIRST`** `- $4326/$4315`
Parameters	None
Purpose	This reads the first anniversary entry into EXTRACTBUF

Routine	**`READ_ANNIV_NEXT`** `- $4335/$4324`
Parameters	None
Purpose	This reads the next anniversary entry into EXTRACTBUF

Scanning support

Routine	**`TEST_SCAN_START`** `- $4346/$4335`
Parameters	SCAN_MONTH - Month, Day, Year of appointment to compare SCAN_DAY SCAN_YEAR TMAPP_MONTH - Current Month, Day, Year

	TMAPP_DAY TMAPP_YEAR
Purpose	Sets 0,SCAN_FLAGS to indicate that the current scan date is out of range.

Routine	**FIX_SCAN_YEAR** - $4371/$4360
Parameters	SCAN_YEAR - Year to be adjusted
Purpose	Adjusts SCAN_YEAR to account for years past 2000

Routine	**TEST_SCAN_END** - $437E/$436D
Parameters	SCAN_MONTH, SCAN_DAY, SCAN_YEAR - Current scan date SCAN_END_MONTH, SCAN_END_DAY, SCAN_END_YEAR - Limit of the scan range
Purpose	Tests to see if the current scan date is past the end range for the scan. If so, it sets 0,SCAN_FLAGS

Routine	**RESTORE_SCAN_YEAR** - $43AE/$439D
Parameters	SCAN_YEAR - Year to be adjusted
Purpose	Restores SCAN_YEAR to be in the 0-99 range (After a call to FIX_SCAN_YEAR)

Routine	**INCREMENT_SCAN_DATE** - $43B9/$43A8
Parameters	SCAN_MONTH, SCAN_DAY, SCAN_YEAR
Purpose	Increments the current scan day by one

Routine	**GET_SCAN_MONTHLEN** - $43E0/$43CF
Parameters	None
Purpose	This computes the end of the month based on SCAN_MONTH and SCAN_YEAR

Routine	**DECREMENT_SCAN_DATE** - $43F4/$43E3
Parameters	SCAN_MONTH, SCAN_YEAR
Purpose	Decrements the scan data by one

Appointment support

Routine	**FIND_APPT_NOW** - $4415/$4404
Parameters	None
Purpose	This finds and reads in an appointment which will occur next after the current time in the current time zone. The appointment is put into EXTRACTBUF and all appropriate variables are set.

Routine	**FIND_APPT_SCAN** - $441B/$440A
Parameters	SCAN_MONTH,DAY,YEAR
Purpose	This finds and reads in an appointment which will occur next after the current scan values. The appointment is put into EXTRACTBUF and all appropriate variables are set.

Routine	**SET_APPTFIND_SCAN** - $4422/$4411
Parameters	SCAN_MONTH, SCAN_DAY, SCAN_YEAR
Purpose	This copies over the current SCAN variables into the APPTFIND variables

Routine	**READ_APPT_NEXT** - $442C/$441B

Parameters	APPT_CURRENT, APPT_LAST - current and last appointment entries
Purpose	This reads in the next appointment into EXTRACTBUF

Routine	**APPT_LATCH_ENTRYDATA** - $4468/$4457 **APPT_LATCH_ENTRYONLY** - $446C/$445B
Parameters	EXTRACTBUF - current appointment entry APPTEST_YEAR - year of the entry
Purpose	These copy the current appointment data into the corresponding system variables The ENTRYONLY routine doesn't copy over the year because it presumably has already been copied.

Routine	**READ_APPT_PREV** - $447C/$446B
Parameters	APPT_CURRENT, APPT_LAST - current and last appointment entries
Purpose	This reads in the previous appointment into EXTRACTBUF

Routine	**FIND_APPT_ENTRY** - $44C6/$44B5
Parameters	APPTFIND_YEAR,DAY,MONTH,QHOUR,HOUR
Purpose	This finds an appointment that matches or exceeds the APPTFIND values

Routine	**APPT_LATCH_ENTDYDATA** - $45A5/$4594
Parameters	APPT_ENTRY - Entry to latch appointment information for
Purpose	This copies the current appointment entry into the corresponding system variables so that we can continue comparing appointments

Routine	**CHECK_APPOINTMENTS** - $45B9/$45A8
Parameters	APPT_QHOUR_NOW - The current quarter-hour APPT_BASEYEAR - The base year for the first appointment
Purpose	This tests to see if any appointments are ready to go off. It posts a nested app for any appointments

Routine	**SET_APPTFIND_NOW** - $462A/$4619
Parameters	None
Purpose	Sets the appointment find variables to the current time

Routine	**READ_APPT_FIRST** - $4686/$4675
Parameters	APPT_FIRST
Purpose	Read in the first appointment

Routine	**READ_APPT_LAST** - $469D/$468C
Parameters	APPT_LAST - the entry of the last appointment
Purpose	This reads in the last appointment entry

Routine	**CHECK_APPT_TIME** - $46B7/$46A6
Parameters	None
Purpose	This checks to see if any appointments are ready to go off

Routine	**READ_APPT_PACKET1** - $473A/$4729
Parameters	None

Purpose	This reads the first appointment packet into EXTRACTBUF

Routine	**READ_NEXT_APPT_PACKET** - $4749/$4738
Parameters	None
Purpose	This reads in the next appointment packet into EXTRACTBUF

Routine	**READ_APPT_CURRENT** - $475A/$4749
Parameters	APPT_CURRENT - the appointment entry to be read
Purpose	This reads in the current appointment entry into EXTRACTBUF

Internal

Routine	**ANNIV_GETMONTHLEN** - $426A/$4259
Parameters	ANNIV_MONTH – Month to calculate ANNIVTEST_YEAR – Year to calculate
Purpose	This computes the number of days in the given month

Routine	**ACQUIRE_TIME** - $4F22/$4F11
Parameters	None
Purpose	This acquires the right to change the time. All alarms and anniversaries will temporarily be ignored until RELEASE_TIME has been called

Routine	**RELEASE_TIME** - $4F2E/$4F1D
Parameters	None
Purpose	This releases the lock on time and allows all alarms and anniversaries to be checked once again.

Indiglo support

Routine	**QUEUE_INDIGLO_OFF** - $49D9/$4C38
Parameters	None
Purpose	Queue up the timer for shutting off the Indiglo if the Indiglo is enabled and we are in night mode.

Routine	**INDIGLO_OFF** - $4E8E/$4E7D
Parameters	None
Purpose	This routine turns off the Indiglo light

Routine	**NIGHTMODE_INDIGLO_ON** - $49E6/$4C45
Parameters	None
Purpose	Queue up the timer for shutting off the Indiglo if the Indiglo is enabled and we are in night mode. The INDIGLO_ON routine just simply turns the Indiglo on immediately

Routine	**INDIGLO_ON** - $49EC/$4C4B
Parameters	None
Purpose	Queue up the timer for shutting off the Indiglo if the Indiglo is enabled and we are in night mode. The INDIGLO_ON routine just simply turns the Indiglo on immediately

Sound Support

Routine	**SNDSTART** - $4E4A/$4E39
Parameters	SYSSOUND - Current sound to be playing
Purpose	Start playing the current sound in SYSSOUND

Routine	**STOP_ALL_SOUND** - $4E68/$4E57
Parameters	None
Purpose	Keep the sound hardware running or reset everything else

Routine	**PLAYCONF** - $4E7A/$4E69
Parameters	None
Purpose	Play a confirmation sound

Routine	**PLAYBUTTON** - $4E80/$4E6F
Parameters	None
Purpose	Play the button beep sound if no other sound is currently playing

Routine	**PLAY_HOURLY** - $4EB1/$4EA0
Parameters	None
Purpose	Plays the hourly sound if nothing else is playing and sounds are enabled

Routine	**SNDSTOP** - $4F3A/$4F29
Parameters	None
Purpose	This stops whatever sound is currently playing

Routine	**PLAY_BUTTON_SAFE** - $4F46/$4F35
Parameters	None
Purpose	This will play the button beep sound if it hasn't just been played

Event support

Routine	**POSTEVENT** - $4E89/$4E78
Parameters	A - Event to be posted.
Purpose	Post a event to the internal processing queue This posts an event to run through the processing loop for the current applet. Typical user events are in the $30-$3F range.

Routine	**CALL_NESTEDAPP** - $4F4D/$4F3C
Parameters	A - Nested application number. This is one of the three defined apps: 9 = APP2_ALARM - Alarm (while another app is running) 10 = APP2_APPT - Appointment (while another app is running) 11 = APP2_WRIST - Wristapp (while another app is running) X - Parameter to pass to the nested application
Purpose	This sets up to call a nested application while the current one is running. Up to 5 apps may be nested (although there are only 3 potential ones defined). If more than 5 have been called the oldest one will be forgotten. When the nested app is called, NESTED_APP will be set to the application number passed in and NESTED_PARM will contain the X parameter passed in

Packet/EEProm Support

Routine	**UNPACK_PHONENUM** - $4FBF/$4FAE
Parameters	EXTRACTBUF+1 – Pointer to 6 bytes of compressed phone number information
Returns	BUF_PHONENUM – Contains 12 byte unpacked number
Purpose	This gets a compressed phone number and puts it in the phone number buffer Phone numbers are compressed into nibbles instead of bytes, allowing a number to be packed in half the space. As a result, a number can contain only 16 possible characters: “01234567890CFHPW “ Any other characters are encoded as a space before being sent down. The presumption is that the characters allow for the number and indicators for: Cell Fax Home Pager Work

Routine	**UNPACK_STRING** - $4FF0/$4FDF
Parameters	PARM_UNPACKOFF - Offset into the start of the compressed buffer EXTRACTBUF - packed data
Returns	MSGBUF - Contains the unpacked string
Purpose	This gets a compressed string and puts it into the scrolling message buffer Strings are packed 6 bits across so that 4 unpacked characters can fit in 3 bytes This routine will unpack enough bits so that the resultant message buffer is exactly 32 bytes long. It is assumed that the end of the buffer message is stored in the packed string.

Routine	**READ_PACKET** - $503E/$502D
Parameters	PARM_LEN - Number of bytes to copy PARM_PACKET – Packet number to read X - Packet group to search (0,2,4,6) 0 = APPT Entries 2 = List entries 4 = Phone Number 6 = Anniversaries
Purpose	Reads the requested packet into EXTRACTBUF

Routine	**FIND_PACKET** - $5044/$5033
Returns	INST_ADDRHI:INST_ADDRLO - points to the start of the packet
Parameters	PARM_PACKET - Packet number to locate X - Packet group to search (0,2,4,6) 0 = APPT Entries 2 = List entries 4 = Phone Number 6 = Anniversaries
Purpose	This advances to the given packet in the packet group

Routine	**DO_TRANSFER** - $505F/$504E
Parameters	PARM_LEN - Number of bytes to copy INST_ADDRHI - Address of source data to copy
Purpose	This transfers the data from the indicated location to EXTRACTBUF The source can be the EEPROM or somewhere else in memory

Routine	**TOGGLE_ENTRYFLAG** - $5077/$5066
Parameters	None

Purpose	This toggles the high bit of the first byte in an entry

Routine	**INIT_EEPROMPOINTERS** - $5080/$506F
Parameters	None
Purpose	Initializes all of the EEProm data pointers to reflect empty data for all of the applications

Routine	**RESET_EEPROMENTRIES** - $508D/$507C
Parameters	None
Purpose	Re-Initializes all of the EEProm data pointers to reflect empty data for all of the applications

Routine	**REINIT_APP_DATA** - $50A7/$5096
Parameters	None
Purpose	This routine is called after new data has been loaded into the EEPROM

INST Support

Routine	**MAKE_INST_LDA** - $50B4/$50A3 **MAKE_INST_LDA_X** - $50B8/$50A7 **MAKE_INST_STA** - $50BC/$50AB
Parameters	None
Purpose	These routines make the INST2 opcodes to be an LDA or STA $nnnn,X instruction

Routine	**ADD_INSTADDR** - $50C7/$50B6
Parameters	A - value to add to the current INST_ADDR base address
Purpose	This takes an offset value and subsumes it into the already constructed instruction starting at INST_OPCODE

Routine	**SET_INSTADDR_0110** - $50D7/$50C6
Parameters	X – R
Purpose	This routine sets INST_ADDRHI:INST_ADDRLO to be 0110

Routine	**GET_INST_BYTE** - $50EB/$50DA
Parameters	INST_ADDRHI:INST_ADDRLO - the pointer to the byte to get
Purpose	This routine gets the single byte from the indicated location either in the EEPROM or somewhere in memory.

Routine	**WRITE_FLAG_BYTE** - $510A/$50F9
Parameters	INST_ADDRHI:INST_ADDRLO - the pointer to the byte to write to
Purpose	This routine writes a single byte to the indicated location either in the EEPROM or somewhere in memory. ?????? This adjusts an address relative to the Sound buffers.
Parameters	A - Offset into the sound data area

Routine	**FILL_EXTRACTBUF** - $513E/$512D
Parameters	PARM_LEN - Number of bytes to be copied INST_ADDRHI:INST_ADDRLO - Address in Prom to read
Returns	EXTRACTBUF – Contains the bytes read in from the EEPROM
Purpose	This copies data from the EEPROM to the EXTRACTBUFF Note that this buffer is only 31 bytes long although this routine can support up to 256 bytes.

Routine	**SAVE_EXTRACTBUF** - $515D/$514C
Parameters	PARM_LEN - Number of bytes to be copied INST_ADDRHI:INST_ADDRLO - Address in Prom to write EXTRACTBUF - Contains the bytes to write to the EEPROM
Purpose	This copies data from the EXTRACTBUFF to the EEPROM Note that this buffer is only 31 bytes long although this routine can support up to 256 bytes.

Routine	**SYSTEM_RESET** - $519B/$518A
Parameters	None
Purpose	This routine is the main reset routine for starting up the watch. It cleans up all of memory and starts the processing once again

Routine	**INIT_SOUNDS** - $5265/$51F2
Parameters	None
Purpose	This routine initializes the default sounds

Routine	**ENABLE_EYE** - $53A6/$5367
Parameters	None
Purpose	This routine enables the received on the watch to download from the screen. It also seems to wait for SERIAL_DATA/SERIAL_CONTROL to settle down

Routine	**DISABLE_EYE** - $53BD/$537E
Parameters	None
Purpose	This disables the eye for normal watch operation

Routine	**SET_SYS_07** - $53C8/$5389
Parameters	None
Purpose	???? This routine resets the SYS_07 hardware

Routine	**CLEAR_SYS_07** - $53CF/$5390
Parameters	None
Purpose	???? This routine resets the SYS_07 hardware Clears 1,HW_FLAGS

Routine	**RESET_SYS_07** - $53D5/$5396
Parameters	None
Purpose	???? This routine resets the SYS_07 hardware

Routine	**INITHW_SYS_07** - $53DC/$539D
Parameters	SYSTEMP2 - 0 or $c1 to indicate how the hardware is to be reset
Purpose	???? This routine initializes the SYS_07 hardware

Routine	**SETHW_07_08_C1** - $53F4/$53B5
Parameters	A - $C1 - Value to be poked into SYS_08
Purpose	???? Resets the SYS_07, SYS_08 hardware. There is a timing loop associated with this reset operation.

Routine	**WRITE_ACQUIRE** - $543C/$542B
Parameters	None
Purpose	This routine acquires the EEPROM for writing. It will also turn off any playing sound as well as the INDIGLO in order to conserve power while doing the writing.

Routine	**WRITE_RELEASE** - $5448/$5437
Parameters	None
Purpose	This routine releases the EEPROM for writing. If the Indiglo had been previously on, it is turned back on.

Routine	**MAKE_INST2_LDA_X** - $5453/$5442 **MAKE_INST2_STA_X** - $5457/$5446
Parameters	None
Purpose	These routines make the INST2 opcodes to be an LDA or STA $nnnn,X instruction

Routine	**PROM_READ** - $5462/$5451
Parameters	INST2_COUNT - Number of bytes to be copied PROM_ADDRHI:PROM_ADDRLO - Address in Prom to read INST2_ADDRHI:INST2_ADDRLO - Address to copy data to
Purpose	This copies data from the EEPROM to the indicated buffer

Routine	**PROM_WRITE** - $5488/$5477
Parameters	INST2_COUNT - Number of bytes to be copied PROM_ADDRHI:PROM_ADDRLO - Address in Prom to write INST2_ADDRHI:INST2_ADDRLO - Address to copy data from
Purpose	This copies data to the EEPROM from the indicated buffer

Routine	**SET_INDIGLO** - $5504/$54F3
Parameters	0,HW_FLAGS – Indicates request for on or off
Purpose	This routine turns on/off the Indiglo light

Scrolling Messages

Routine	**PUTSCROLLMSG** - $5522/$5511
Parameters	MSGBUF - the message to scroll terminated by SEPARATOR
Purpose	Initialize a scrolling message

Routine	**SCROLLMSG** - $5545/$5534
Parameters	MSGBUF - Message to be scroll terminated by a SEPARATOR character
Purpose	Start the scrolling cycle for the current message

Routine	**SCROLLMSG_CONT** - $5549/$5538
Parameters	MSGBUF - Message to be scroll terminated by a SEPARATOR character SCROLL_TICS - The current tic count in the cycle
Purpose	Start the scrolling cycle for the current message, but don't reset the scrolling cycle wait count.

Blinking routines

Routine	**START_BLINKX** - $55BB/$55AA

Parameters	A - Blinking function to be selected		
	0	BLINK_YEAR	Blink the year in the right place according to the current time format
	1	BLINK_SECONDS	Blink two characters point to by UPDATE_PARM on the right two digits of the middle line - Used for the seconds
	2	BLINK_AMPM	Blink AM/PM on the right most digits of the middle line (A or P pointed to by UPDATE_PARM)
	3	BLINK_MONTH	Blink the month in the right place according to the current time format
	4	BLINK_HMONTH	Blink the month in the right place according to the current time format for a half date (no year)
	5	BLINK_DAY	Blink the day in the right place according to the current time format
	6	BLINK_HDAY	Blink the day in the right place according to the current time format for half dates
	7	BLINK_MID12	Blink the left two blank padded digits on the middle line (value pointed to by UPDATE_PARM)
	8	BLINK_HOUR	Blink the Hour (left two segments on the middle line) and AM/PM indicator (hour point to by UPDATE_PARM)
	9	BLINK_MID34	Blink the middle two zero padded digits on the middle line (value pointed to by UPDATE_PARM)
	10	BLINK_SEGMENT	Blink a single segment indicated by UPDATE_POS and mask in UPDATE_VAL
	11	BLINK_DIGIT	Blink solid black cursor for the digit (UPDATE_POS is the location on the bottom line)
	12	BLINK_TZONE	Blink the timezone information (Pointed to by UPDATE_PARM)
	13	BLINK_TOP34	Blink the middle zero padded two digits on the top line (value pointed to by UPDATE_PARM)
	X - single byte parameter to the particular blinking function		
Purpose	Establish and call the specified blinking routine		

Routine	**START_BLINKP** - $55BF/$55AE		
Parameters	A - Blinking function to be selected		
	0	BLINK_YEAR	Blink the year in the right place according to the current time format
	1	BLINK_SECONDS	Blink two characters point to by UPDATE_PARM on the right two digits of the middle line - Used for the seconds
	2	BLINK_AMPM	Blink AM/PM on the right most digits of the middle line (A or P pointed to by UPDATE_PARM)
	3	BLINK_MONTH	Blink the month in the right place according to the current time format
	4	BLINK_HMONTH	Blink the month in the right place according to the current time format for a half date (no year)
	5	BLINK_DAY	Blink the day in the right place according to the current time format
	6	BLINK_HDAY	Blink the day in the right place according to the current time format for half dates
	7	BLINK_MID12	Blink the left two blank padded digits on the middle line (value pointed to by UPDATE_PARM)
	8	BLINK_HOUR	Blink the Hour (left two segments on the middle line) and AM/PM indicator (hour point to by UPDATE_PARM)

	9	BLINK_MID34	Blink the middle two zero padded digits on the middle line (value pointed to by UPDATE_PARM)
	10	BLINK_SEGMENT	Blink a single segment indicated by UPDATE_POS and mask in UPDATE_VAL
	11	BLINK_DIGIT	Blink solid black cursor for the digit (UPDATE_POS is the location on the bottom line)
	12	BLINK_TZONE	Blink the timezone information (Pointed to by UPDATE_PARM)
	13	BLINK_TOP34	Blink the middle zero padded two digits on the top line (value pointed to by UPDATE_PARM)
	X - Address of parameter to the particular blinking function		
Purpose	Establish and call the specified blinking routine		

Update functions

Routine	`START_UPDATEX - $57C3/$56C4`		
Parameters	A - Update function to be selected		
	0	UPD_YEAR	Update the year
	1	UPD_MONTH	Update the Month
	2	UPD_HMONTH	Update the Month in Half date format
	3	UPD_DAY	Update the day
	4	UPD_HDAY	Update the day in half date format
	5	UPD_MID12	Update MID12
	6	UPD_HOUR	Update the hour
	7	UPD_MID34	Update MID34
	8	UPD_DIGIT	Update the digit at UPDATE_POS
	X - single byte parameter to the particular update function		
Purpose	Establish and call the specified update function		

Routine	`START_UPDATEP - $57C7/$56C8`		
Parameters	A - Update function to be selected		
	0	UPD_YEAR	Update the year
	1	UPD_MONTH	Update the Month
	2	UPD_HMONTH	Update the Month in Half date format
	3	UPD_DAY	Update the day
	4	UPD_HDAY	Update the day in half date format
	5	UPD_MID12	Update MID12
	6	UPD_HOUR	Update the hour
	7	UPD_MID34	Update MID34
	8	UPD_DIGIT	Update the digit at UPDATE_POS
	X - Pointer to parameters for the update function		
Purpose	This establishes an update function. Update functions are called every 8/10th of a second. This function will update a number in an upward or downward direction based on the setting of 0,SYSFLAGS		

Format Routines

These routines are useful for formatting numbers into the corresponding character representation.

Routine	**FMTXLEAD0** - $593E/$583F
Parameters	X - value to be formatted. 0-9 results in 0 followed by the digit 10-99 results in number for both digits
Purpose	Formats into DATDIGIT1/2 with leading zeros

Routine	**FMTBLANK0** - $594D/$584E
Parameters	X - value to be formatted. 0 results in all blanks. 1-9 results in blank followed by the digit 10-99 results in number for both digits
Purpose	Formats a number into DATDIGIT1/2

Routine	**FMTX** - $5951/$5852
Parameters	X - value to be formatted. 0-9 results in blank followed by the digit 10-99 results in number for both digits
Purpose	Formats a number into DATDIGIT1/2

Routine	**FMTSPACE** - $595C/$585D
Parameters	None
Purpose	This routine simply puts spaces into DATDIGIT1 DATDIGIT2

Routine	**FMTBLANK0B** - $5963/$5864
Parameters	X - value to be formatted. 0 results in all blanks. 1-9 results in blank followed by the digit 10-99 results in number for both digits
Purpose	Formats a number into DATDIGIT1/2. This routine does not appear to be used anywhere and seems to do exactly the same thing as FMTBLANK0

Routine	**FIXLEAD0** - $5A2A/$592B
Parameters	None
Purpose	If the first digit is a zero, replace it with a blank

Line routines

These routines are useful for putting strings on the display

<table>
<tr><td>Routine</td><td colspan="9"><b>PUTLINE3</b> - $56D5/$59E7</td></tr>
<tr><td rowspan="4">Parameters</td><td colspan="8">A = Position</td><td></td></tr>
<tr><td>S1</td><td>S2</td><td>S3</td><td>S4</td><td>S5</td><td>S6</td><td>S7</td><td>S8</td><td rowspan="2"></td></tr>
<tr><td>$47</td><td>$3D</td><td>$33</td><td>$27</td><td>$1D</td><td>$13</td><td>$09</td><td>$0a</td></tr>
<tr><td colspan="8">X = Character in Timex ASCII to display</td><td></td></tr>
<tr><td>Purpose</td><td colspan="9">Put a single character on the bottom line of the display This routine pokes in a single digit on the display. Note that the last digit is backwards and upside down in the hardware.</td></tr>
</table>

Routine	**PUTLINE1** - $570D/$5A33

Parameters	A = Position						
	T1	T2	T3	T4	T5	T6	
	$46	$3E	$34	$2C	$22	$14	
	X = Character in Timex ASCII to display						
Purpose	Put a single character on the top line of the display						

Routine	**PUTLINE2** - $5745/$5A6B						
Parameters	A = Position						
	M1	M2	M3	M4	M5	M6	
	$46	$3E	$34	$2C	$22	$14	
	X = Character in Timex ASCII to display						
Purpose	Put a single character on the second line of the display						

Routine	**SETALL** - $5776/$5A9C
Parameters	None
Purpose	Turns on all segments on the entire display

Routine	**CLEARALL** - $577A/$5AA0
Parameters	None
Purpose	Clear the entire display

Routine	**CLEARBOT** - $5787/$5AAD
Parameters	None
Purpose	Clear the bottom line of the display

Routine	**CLEAR_RANGE** - $5793/$5ABF
Parameters	A – Initial offset to be clearing from X – Number of words to clear
Purpose	Turn off all bits on the display at the given offsets

Routine	**CLEARSYM** - $579F/$5ACB
Parameters	None
Purpose	Turns off all the non digit symbols segments (including dots, dashes and colons)

Routine	**BANNER8** - $5845/$5746
Parameters	A = Offset from 0110 for the start of an 8 character Timex string
Purpose	Display an 8 character string

Routine	**PUTMSGXBOT** - $5849/$574A
Parameters	A = Message selector number. Valid values from 0 to 27. They correspond to the same strings passed into PUTMSGBOT scaled down by 8
Purpose	Display an 8 character system string on the bottom line

Routine	**PUTMSGBOT** - $584C/$574D

Parameters	A = Offset into message selector string. Valid values from $00 to $d8 at 8 Byte offsets. $E0 is the start of the 6 byte top/mid message strings.		
	$00	SYS8_MON	`“MON     “`
	$08	SYS8_TUE	`“TUE     “`
	$10	SYS8_WED	`“WED     “`
	$18	SYS8_THU	`“THU     “`
	$20	SYS8_FRI	`“FRI     “`
	$28	SYS8_SAT	`“SAT     “`
	$30	SYS8_SUN	`“SUN     “`
	$38	SYS8_VERDATE	`“ 802003 “`
	$40	SYS8_VERSION	`“  V2.0  “`
	$48	SYS8_MODE	`“  MODE  “`
	$50	SYS8_SET_MODE	`“SET MODE”`
	$58	SYS8_SET	`“SET     “`
	$60	SYS8_TO	`“TO      “`
	$68	SYS8_FOR	`“FOR     “`
	$70	SYS8_ENTRIES	`“ENTRIES “`
	$78	SYS8_UPCOMING	`“UPCOMING”`
	$80	SYS8_ENTRY	`“ ENTRY  “`
	$88	SYS8_SCAN	`“ SCAN   “`
	$90	SYS8_SCAN_RIGHT	`“    SCAN”`
	$98	SYS8_SYNCING	`“ SYNCING”`
	$a0	SYS8_PROGRESS	`“PROGRESS”`
	$a8	SYS8_DATA_OK	`“ DATA OK”`
	$b0	SYS8_RESEND	`“-RESEND-“`
	$b8	SYS8_ABORTED	`“ ABORTED”`
	$c0	SYS8_MISMATCH	`“MISMATCH”`
	$c8	SYS8_SPLIT	`“ SPLIT  “`
	$d0	SYS8_START	`“>=START “`
	$d8	SYS8_STOP	`“>=STOP  “`
Purpose	Display an 8 character system string on the bottom line		

Routine	**PUTDOWTOP** - $5872/$5773
Parameters	X - Day of week (0-6)
Purpose	Displays the two character representation of the day of the week in the upper left of the display

Routine	**PUT6TOP** - $587E/$577F
Parameters	A = Offset from WRIST_MAIN for the start of a 6 byte data item to be put on the top line of the screen. This uses a different encoding for characters where: we have 32 different values which correspond to: `0123456789ABCDEFGH:LMNPRTUWYr -+` e.g. $12=’:’, $13=’L’. It appears that things wrap when you get to $20
Purpose	Display a 6 character string on the top line

Routine	**PUTMSG1** - $5882/$5783

Parameters	A = Offset into message selector string. Valid values from $00 to $a8 at 6 Byte offsets.		
	$00	SYS6_SET	“ SET “
	$06	SYS6_HOLDTO	“HOLDTO”
	$0C	SYS6_ALARM	“ALARM “
	$12	SYS6_ENTER	“ENTER “
	$18	SYS6_HR	“ HR”
	$1E	SYS6_SWITCH	“SWITCH”
	$24	SYS6_TIME	“ TIME “
	$2A	SYS6_FORMAT	“FORMAT”
	$30	SYS6_DAILY	“DAILY “
	$36	SYS6_APPT	“ APPT “
	$3c	SYS6_NO	“ NO “
	$42	SYS6_APPTS	“APPTS “
	$48	SYS6_END_OF	“END OF”
	$4e	SYS6_LIST	“ LIST “
	$54	SYS6_DELETE	“DELETE”
	$5a	SYS6_ANN	“ ANN “
	$60	SYS6_PHONE	“PHONE “
	$66	SYS6_DONE	“ DONE “
	$6c	SYS6_PRI	“PRI “
	$72	SYS6_COMM	“ COMM “
	$78	SYS6_READY	“READY “
	$7e	SYS6_IN	“ IN “
	$84	SYS6_ERROR	“ERROR “
	$8a	SYS6_CEASED	“CEASED”
	$90	SYS6_PC	“PC- “
	$96	SYS6_WATCH	“WATCH “
	$9c	SYS6_CHRONO	“CHRONO”
	$A2	SYS6_TIMER	“TIMER “
	$a8	SYS6_000000	“000000”
Purpose	Display an 6 character system string on the top line		

Routine	**PUT6MID** - $58A8/$57A9
Parameters	A = Offset from WRIST_MAIN for the start of a 6 byte data item to be put on the top line of the screen. This uses a different encoding for characters where: we have 32 different values which correspond to: `0123456789ABCDEFGH:LMNPRTUWYr -+` e.g. $12=’:’, $13=’L’. Beyond $20 you get random junk.
Purpose	Display a 6 character string on the second line

Routine	**PUTMSG2** - $58AC/$57AD
Parameters	A = Offset into message selector string. Valid values from $00 to $a8 at 6 Byte offsets and the strings are the same as for PUTMSG1
Purpose	Display an 6 character system string on the top line

Routine	**CLEARTOP** - $58D2/$57D3
Parameters	None
Purpose	Puts blanks into all 6 top digits (Blanks out the top line)

Routine	**CLEARMID** - $58D8/$57D9
Parameters	None
Purpose	Puts blanks into all 6 Middle digits (Blanks out the middle line)

Routine	**CLRTOP12** - $58DE/$57DF
Parameters	None
Purpose	Puts blanks into top Digits 1 and 2

Routine	**PUTTOP12** - $58E0/$57E1
Parameters	None
Purpose	Puts DATDIGIT1/2 into TOP Digits 1 and 2

Routine	**CLRTOP34** - $58EE/$57EF
Parameters	None
Purpose	Puts blanks into TOP Digits 3 and 4

Routine	**PUTTOP34** - $58F0/$57F1
Parameters	None
Purpose	Puts DATDIGIT1/2 into TOP Digits 3 and 4

Routine	**CLRTOP56** - $58FE/$57FF
Parameters	None
Purpose	Puts blanks into TOP Digits 5 and 6

Routine	**PUTTOP56** - $5900/$5801
Parameters	None
Purpose	Puts DATDIGIT1/2 into TOP Digits 5 and 6

Routine	**CLRMID12** - $590E/$580F
Parameters	None
Purpose	Puts blanks into Middle Digits 1 and 2

Routine	**PUTMID12** - $5910/$5811
Parameters	None
Purpose	Puts DATDIGIT1/2 into Middle Digits 1 and 2

Routine	**CLRMID34** - $591E/$581F
Parameters	None
Purpose	Puts blanks into Middle Digits 3 and 4

Routine	**PUTMID34** - $5920/$5821

Parameters	None
Purpose	Puts DATDIGIT1/2 into Middle Digits 3 and 4

Routine	**`CLRMID56`** `- $592E/$582F`
Parameters	None
Purpose	Puts blanks into Middle digits 5 and 6

Routine	**`PUTMID56`** `- $5930/$5831`
Parameters	None
Purpose	Puts DATDIGIT1/2 into Middle Digits 5 and 6

Routine	**`SAYEOLMSG`** `- $5979/$587A`
Parameters	None
Purpose	Puts 'END OF LIST' on the display

Routine	**`SAYHOLDTODELETE`** `- $598A/$588B`
Parameters	None
Purpose	Puts 'HOLD TO DELETE ENTRY' on the display

Routine	**`PUT_PHONENUM`** `- $59A2/$58A3`
Parameters	None
Purpose	Puts a phone number on the top two lines of the display (Up to 12 digits). If there is a non blank character as the third digit, a - is turned on between the 3rd and 4th digits to separate out what is presumably the area code

Routine	**`PUTYEARMID`** `- $59D9/$58DA`
Parameters	X - Year to be formatted on the display
Purpose	Puts the current year on the right half of the middle display. If the year passed in is less than 50, it is assumed to be 20xx, above 50 it is processed as 19xx giving a range of 1950-2049

Routine	**`CLEAR_HMONTH`** `- $59F8/$58F9`
Parameters	None
Purpose	blank out the 2 character day for a half date (no year) based on the current time zone date format

Routine	**`PUT_HMONTHX`** `- $59FD/$58FE`
Parameters	X - Day to be displayed
Purpose	Put the leading space 2 digit month in the appropriate spot on the display based on the current time zone date format for a half date (no year)

Routine	**`CLEAR_HDAY`** `- $5A11/$5912`
Parameters	None
Purpose	blank out the 2 character day for a half date (no year) based on the current time zone date format

Routine	**`PUT_HDAYX`** `- $5A16/$5917`
Parameters	X - Day to be displayed
Purpose	Put the leading zero 2 digit day in the appropriate spot on the display based on the current time

	zone date format for a half date (no year)

Routine	**CLEAR_MONTH** - $5A36/$5937
Parameters	None
Purpose	blank out the 2 character month based on the current time zone date format

Routine	**CLEAR_DAY** - $5A4F/$5950
Parameters	None
Purpose	blank out the 2 character day based on the current time zone date format

Routine	**PUTBOT678** - $5A86/$5987
Parameters	X - Pointer to 3 byte location containing bytes to put on the display (pointed to by x) 3 bytes in TIMEX ASCII. Because the X register is used to index to them, they must be located in the first 256 bytes of memory.
Purpose	Puts three digits into the lower corner of the display. Typically this is the time zone information.

Routine	**CLEAR_YEAR** - $5A6F/$5970
Parameters	None
Purpose	blank out the 2 character year based on the current time zone date format

Routine	**IPUT_MONTHX** - $5A3B/$593C
Parameters	X - Month to be displayed
Purpose	Put the leading space 2 digit month in the appropriate spot on the display based on the current time zone date format

Routine	**IPUT_DAYX** - $5A54/$5955
Parameters	X - Day to be displayed
Purpose	Put the leading zero 2 digit day in the appropriate spot on the display based on the current time zone date format

Routine	**IPUT_YEARX** - $5A74/$5975
Parameters	X - Year to be displayed
Purpose	Put the leading zero 2 digit year in the appropriate spot on the display based on the current time zone date format

Routine	**PUTHALFDATESEP** - $5AA0/$59A1
Parameters	None
Purpose	Show the separator character for a half date (no year) based on the current date format

Routine	**PUTDATESEP** - $5AAB/$59AC
Parameters	None
Purpose	Show the separator characters for a full date based on the current date format

Routine	**PUT_LETTERX** - $5ACE/$59CF
Parameters	A - Character to be displayed X - Offset on the bottom line to put character

Purpose	Put a single character at the appropriate spot on the bottom line

Routine	**`PUT_HOURX`** `- $5AD9/$59DA`
Parameters	X - Hour to be displayed
Purpose	Put the hour on the first two digits of the middle line along with the colon

Routine	**`UPDATE_SECONDS`** `- $625E/$6267`
Parameters	None
Purpose	This routine checks the current TIC count and updates the seconds based on that TIC. If the minute rolls over, we also set the flags so that the rest of the system can respond to it.

Routine	**`SHOW_TIME_DISPLAY`** `- $676A/$6773`
Parameters	None
Purpose	Display the time information based on the current time zone and whether or not we might be in time set mode. All symbols are updated

Routine	**`PUT_YEARX`** `- $67CC/$67D5`
Parameters	X - Year to be displayed
Purpose	Put the leading zero 2 digit year in the appropriate spot on the display based on the current time zone date format

Routine	**`PUT_MONTHX`** `- $67D0/$67D9`
Parameters	X - Month to be displayed
Purpose	Put the leading space 2 digit month in the appropriate spot on the display based on the current time zone date format

Routine	**`PUT_DAYX`** `- $67D4/$67DD`
Parameters	X - Day to be displayed
Purpose	Put the leading zero 2 digit day in the appropriate spot on the display based on the current time zone date format

Routine	**`SAY_HOURX`** `- $67D8/$67E1`
Parameters	X - Hour to be displayed
Purpose	Puts up the hour on the display along with an AM/PM indicator and a Colon. This code respects the current 12/24 hour format.

Routine	**`CLEAR_PM`** `- $6815/$681C`
Parameters	NONE
Purpose	Turn off the PM indicator.

Routine	**`CLEAR_AM`** `- $681C/$6825`
Parameters	NONE
Purpose	Turn off the AM indicator.

Routine	**`PUT_MINUTEX`** `- $6823/$682C`
Parameters	X - minute (0-59) to be displayed

Purpose	This puts the minute in the middle two digits on the middle line followed by a period

Routine	**SHOWSEC_TENS** - $6830/$6839
Parameters	SECOND_TENS - Value to be put on the display
Purpose	Puts the character at SECOND_TENS onto the next to the last digit on the middle line

Routine	**SHOWSEC_ONES** - $6838/$6841
Parameters	SECOND_ONES – Value to be put on the display
Purpose	Puts the character at SECOND_ONES onto the last digit on the middle line

Routine	**SHOWNIGHT_SYM** - $6840/$6849
Parameters	None
Purpose	Displays the night symbol if we are in night mode

Routine	**SAY_HOLD_TO** - $6855/$685E
Parameters	None
Purpose	Puts 'HOLD-TO' on the top line

Routine	**FIX_TMAPP_DAY** - $6861/$686A
Parameters	None
Returns	A - limited day of the month
Purpose	Based on TMAPP_MONTH, TMAPP_YEAR, this routine limits the day of the month to a legal one

Routine	**TMAPP_COPYTZ1** - $6881/$688A
Parameters	None
Purpose	Copies the Hour, Minute, Month, Day, and Year information for Time Zone 1 to the corresponding TMAPP variables.

Routine	**TMAPP_COPYTZ2** - $688C/$6895
Parameters	None
Purpose	Copies the Hour, Minute, Month, Day, and Year information for Time Zone 2 to the corresponding TMAPP variables.

Routine	**GETTZNAME** - $6897/$68A0
Parameters	None
Returns	X - Pointer to the 3 character name of the current time zone

Routine	**GET_MONTHDAYX** - $689F/$68A8
Parameters	X - pointer to two byte location to retrieve Month and Day
Returns	A - The current year for the current time zone
Purpose	Returns the year for the current time zone

Routine	**GET_YEAR** - $68B2/$68BB
Parameters	None

Returns	A - The current year for the current time zone
Purpose	Returns the year for the current time zone

Routine	**GET_HOURFORMAT** - $68BB/$68C4
Parameters	None
Returns	X - 12 or 24 depending on the time format
Purpose	Returns the 12/24 hour time format for the current time zone

Routine	**GET_DATEFMT** - $68CB/$68D4
Parameters	None
Returns	A - Date format mask for the current time zone One of: 0 = DATEFMT_MMDDYY = Date Format is MM-DD-YY 1 = DATEFMT_DDMMYY = Date Format is DD-MM-YY 2 = DATEFMT_YYMMDD = Date Format is YY-MM-DD and One of 0 = DATEFMT_SEPDASH = Dates are separated by dashes 4 = DATEFMT_SEPDOTS = Dates are separated by periods
Purpose	Returns the date format for the current time zone

Routine	**CALC_DOW_X** - $68D5/$68DE
Parameters	X - Pointer to Month, Day, Year block
Purpose	Computes the Day of the Week from the Month, Day, Year information

Routine	**COPY_MDY** - $68DB/$68E4
Parameters	X - pointer to Month, Day, Year block to copy
Purpose	Copies over the Month, Day, and Year information in preparation for calling CALC_DOW

Routine	**ACQUIRE** - $68E8/$68F1
Parameters	None
Purpose	Disable interrupts for a short piece of code

Routine	**RELEASE** - $68F2/$68FB
Parameters	None
Purpose	Reenable interrupts

Routine	**GET_MONTHLEN** - $68F9/$6902
Parameters	PARM_MONTH, PARM_YEAR contain the month and year to look for
Returns	A - Number of days in the month
Purpose	Computes the number of days in a given month

Routine	**CHECK_TZ** - $690E/$6917
Parameters	None
Purpose	Determine which time zone is to be displayed. Carry flag clear = TZ1 Carry flag set = TZ2

Routine	**`CALC_DOW`** `- $691C/$6925`
Parameters	CURRENT_MONTH, CURRENT_DAY, CURRENT_YEAR - holds the information to calculate from
Returns	A - Day of Week (0=Monday...6=Sunday)
Purpose	Calculates the day of the week from the given information

Routine	**`LIST_DISPLAY_CURRENT`** `- $6ABB/$6AC4`
Parameters	None
Purpose	Display the current list entry. List entries are up to 31 bytes long with Byte 0: Completion status. Negative numbers indicate that it is not yet done Byte 1: The priority of the event. 0 indicates no priority Bytes 2-26: The packed text of the message (Up to 32 bytes unpacked) Bytes 27-31 – Wasted since they can never be unpacked

Routine	**`INCA_WRAPX`** `- $6B0D/$6B16`
Parameters	A - Number to be incremented X - Range to hold number within
Purpose	Advance to the next value wrapped within a range

Routine	**`DELAY_X`** `- $6B31/$6B3A`
Parameters	X - Delay interval (Measured in ?) - Note that 1 is the only value ever passed in here
Purpose	Delay for a fixed amount of time

Routine	**`DELAY_X16`** `- $6B43/$6B4C`
Parameters	X - interval to delay for ($C8 is the only value ever passed in)
Purpose	Delay for a fixed amount of time

Routine	**`GETBCDHI`** `- $6B52/$6B5B`
Parameters	X - Hex value to be converted (Range 0-99)
Returns	A - High byte of number in Timex ASCII

Routine	**`GETBCDLOW`** `- $6B5A/$6B63`
Parameters	X - Hex value to be converted (Range 0-99)
Returns	A - Low byte of number in Timex ASCII

Routine	**`ALARM_CHECK`** `- $6BC4/$6C9C`
Parameters	None
Purpose	This routine is called once a minute to check for and raise any alarms

Routine	**`SHOWNOTE_SYM`** `- $6C62/$6C56`
Parameters	None
Purpose	Displays the NOTE symbol if there is a note to be displayed

Routine	**`SHOWALARM_SYM`** `- $6C76/$6C6A`
Parameters	None
Purpose	Displays the ALARM symbol if there are any enabled alarms which are not masked This will also start the alarm symbol blinking if we are in alarm backup mode

Routine	**ALARM_DISPLAY_CURRENT** - $6EF4/$6EFD
Parameters	None
Purpose	Display the current alarm information on the entire display. Daily is put on the top line and the NOTE/ALARM symbols are displayed accordingly

Routine	**ALARM_SHOW_HOURLYNOTE** - $6F39/$6F42
Parameters	None
Purpose	Set the note symbol to the state of the hourly chimes

Routine	**ALARM_SHOW_ALARMSYM** - $6F4A/$6F53
Parameters	ALARM_FLAGS – status of alarm to show
Purpose	Set the alarm symbol to the state of the current alarm

Routine	**ALARM_SHOW_AMPM** - $6F5B/$6F64
Parameters	ALARM_FLAGS - indicates whether a 12 hour format is in AM or PM
Purpose	Set the alarm symbol to the state of the current alarm

Routine	**MASK_ALARMS** - $6FF3/$6FFC
Parameters	None
Purpose	This temporarily disables all alarms by turning on the mask bit (0x02) for all five alarms.

Routine	**UNMASK_ALARMS** - $7000/$7009
Parameters	None
Purpose	This reenables all alarms by turning off the mask bit (0x02) for all five alarms.

Routine	**ANNIV_SHOW_DATE** - $7184/$718D
Parameters	None
Purpose	Displays date for the current anniversary entry

Routine	**ANNIV_SHOW_SCAN_DATE** - $719F/$71A8
Parameters	None
Purpose	Displays date for the current anniversary scan date

Routine	**ANNIV_SHOW_CURRENT** - $71AC/$71B5
Parameters	None
Purpose	Displays the current anniversary entry

Routine	**SHOWREMIND_SYM** - $71D6/$71DF
Parameters	None
Purpose	Displays the reminder symbol if there are any anniversaries within this week. If one is today, this will toggle the remind symbol each time this routine is called

Routine	**OFFREMIND_SYM** - $71EE/$71F7
Parameters	None
Purpose	Turns off the reminder symbol

Routine	**SAY_NO_ANN_ENTRIES** - $71F5/$71FE
Parameters	None
Purpose	Displays the message NO ANN ENTRIES on the display

Routine	**APPT_SHOW_TIME** - $73D7/$73E0
Parameters	SCAN_QHOUR – the quarter hour to display
Purpose	This shows the appointment time on the display (including AM/PM indicator)

Routine	**APPT_SHOW_DATE** - $7439/$7442
Parameters	SCAN_MONTH,SCAN_DAY
Purpose	This shows the appointment date on the display (including the day of the week)

Routine	**APPT_SHOW_SCAN** - $7454/$745D
Parameters	SCAN_MONTH,SCAN_DAY
Purpose	This shows the scan date on the display (including the day of the week) with the year and a message indicating that we are scanning

Routine	**APPT_SHOW_CURRENT** - $7461/$746A
Parameters	None
Purpose	This shows the next upcoming appointment (if any)

Routine	**APPT_SHOW_UPCOMING** - $748E/$7497
Parameters	None
Purpose	This shows the next upcoming appointment (if any)

Routine	**SAY_NO_APPT_ENTRIES** - $74BD/$74C6
Parameters	None
Purpose	This puts NO APPT ENTRIES on the display

Routine	**COMM_CHECK_CRC** - $7C56/$7C3C
Parameters	None
Returns	A - 0 CRC for the current packet matched $ff - CRC for the current packet did not match
Purpose	Compute and validate a CRC for the current packet

Installing a Wristapp

Many people have asked how to install a Wristapp and download it to your watch. While there are people who are using their DataLink with many different operating systems, these instructions only work for the Timex Data Link software for Windows (what comes on the floppy disk with the watch). Note that this is different than Schedule+ or another PIM downloading to the watch.

1. Locate the directory where the DataLink software is installed. Typically this will be C:\Datalink or C:\Program Files\DataLink. In that directory will be a file called TimexDL.DAT

2. Using your favorite editor (Notepad will work just fine), bring in that file to edit.

3. Search in the file for the [WristApps] section. It will consist of several lines like:

```
[WristApps]
WristAppTotal=10
SelectedWristApp=9
WristAppSendOption=True
WristApp000=HEXDUMP0.ZAP
WristApp001=Melody17.ZAP
WristApp002=HELLO.ZAP
WristApp003=NUMBER.ZAP
WristApp004=Update.ZAP
WristApp005=Flash.ZAP
WristApp006=passwd.ZAP
WristApp007=dayfind.ZAP
WristApp008=testsnd.ZAP
WristApp009=endoff.ZAP
```

4. Note the number in the WristAppTotal and increment it by one. (In this case I would change the 10 to an 11)

5. Go to the last entry and add a new line just like the ones above it, but increment the WristApp number by one. In this case, I would add a line after the WristApp009=endoff.ZAP and call that line WristApp010=. Put the name of the wristapp (don't forget the .ZAP extension) on the line. In my example, it would look like:

```
[WristApps]
WristAppTotal=11
SelectedWristApp=9
WristAppSendOption=True
WristApp000=HEXDUMP0.ZAP
WristApp001=Melody17.ZAP
WristApp002=HELLO.ZAP
WristApp003=NUMBER.ZAP
WristApp004=Update.ZAP
WristApp005=Flash.ZAP
WristApp006=passwd.ZAP
WristApp007=dayfind.ZAP
WristApp008=testsnd.ZAP
WristApp009=endoff.ZAP
WristAPP010=NewApp.ZAP
```

6. Save the file

7. Copy the .ZAP file into the APP subdirectory of the DataLink software and you are done.
8. Load up the Datalink Software, and click on the WristApps button.
9. Scroll to the bottom of the list to see your new WristApp
10. Select the wristapp and make sure that the bottom says to send the selected WristApp
11. Select OK and then proceed to download to your watch with the normal COMM mode
12. Enjoy!

My Wristapps

The wristapps that I have written so far. Everything here works for both the 150 and the 150s.

- *TipCalc* - Calculates 10, 15, 20% tips. Thanks to David M. Schreck <dschreck@csfbg.csfb.com> for the idea!
- *Hello* - Tutorial #1 - Hello World! (Now where is my C Compiler?)
- *Number* - Tutorial #2 - Change a single number
- *Update* - Tutorial #3 - Update a number using a system routine
- *Flash* - Tutorial #4 - Blinks and changes the number.
- *Passwd* - Tutorial #5 - Blinks, changes, and selects numbers.
- *DayFind* - Tutorial #6 - gives you the day of the week
- *Sound Test* - Tutorial #7 - Plays one of the 14 possible tones on the watch.
- *EndOff* - Tutorial #8 - Turn off alarms on the weekend
- *HexDump* - Tutorial #9 - Dump out memory.
- *PromDump* - Tutorial #10 - Dump out the contents of the EEPROM.
- *SpendWatch* - Tutorial #11 - Track how much you spend in a day.
- *Sound1* - Tutorial #12 - Create a simple soundscape.
- *3Ball* - Tutorial #13 - Can't make up your mind? Let 3Ball help you out. Thanks to *Wayne Buttles* <timex@fdisk.com>
- *ShipBell* - Tutorial #14 - Beeps on the hour with the number of hours past a shift change. (suggested by *"Theron E. White, CPA"* <twhite@mercury.peganet.com>).
- *Data Hider* - This works for both the 150 and the 150s.
- *Segment Setter* - This allows you to set all of the segments on the display on/off.

Other People's Wristapps

It is wonderful to now see other people creating Wristapps.

- *NumPad* - Michael Polymenakos <mpoly@panix.com> has created an excellent app which has two functions in one. In his own words: "The first thing I miss from my old (and now non-functional) Casio is the ability to record a number quickly when pen and paper are not available. I wrote a small wristapp, NUMPAD, to let me record a 12 digit number... Any comments will be appreciated (especially on replacing the ugly cursor with a 'blink' function that blinks only one digit at a time)." He has also incorporated a chronometer wristapp in with the app to give you two apps in one.

- *3Ball* - Wayne Buttles <timex@fdisk.com> created the first version of this fun app. It's been updated here as a tutorial.

Plans for Wristapps

The wristapps that I plan to create and know everything necessary to create them.

- *WestMinister Chimes* - For that 'Big Ben' sound. With thanks to Pigeon for the sound scheme to make it possible.

Other wristapps that have been suggested (their original comments are presented. I also include my comments in blue).

- *Falling Blocks* - I have been thinking about this for a while. There is really no reason that you can't design a game to take advantage of the segments to do a simple falling-blocks-like game. You would have to turn the display sideways to play it.

- *Slots* - I have also wanted to do this game for a while. The basic idea is to have a slot machine in the watch where you can press a button and take a whirl. The watch should keep track of your winnings. Because of the way the segments are organized, I believe that you can even do a good imitation of the wheels spinning.

- *Dumper* - We need to have a good application that allows the Datalink to talk back to the PC. The obvious way here will be to use the sounds on the watch and listen to them with the SoundBlaster on the PC. Right now the only thing holding us back is someone to create the PC end to listen. I have everything necessary to generate the tones in a predictable manner.

- *Phone Dialer* - The Datalink is just screaming for this application that has been suggested by many people. It is not clear that this is beyond the capabilities of the DataLink, but so far I have only been able to emit the 14 basic tones in the watch. From my understanding of the watch and the hardware, I haven't completely ruled this out as a possibility.

- *Info entry* - "One of the reasons I like the DataLink is because it DOESN'T have an ugly 12 button keypad on it, but I have to admit, it would be nice to be able to enter a phone number when needed. Granted, it would cumbersome to enumerate the desired digits, but I think it would still be useful (could also be used to enter the section # of a large parking lot that you left your car)." *David M. Schreck* <dschreck@csfbg.csfb.com>. This is certainly doable, but it does have some issues to be considered in dealing with the EEProm. See the **EEProms** information to understand why.

- *Screen Saver* - "Not in the true sense of the phrase, of course, and this one you would have to purposely invoke. I imagine that those who are artistically inclined might think up a creative and interesting way to cycle through the available display fields." *David M. Schreck* <dschreck@csfbg.csfb.com>. If someone proposes a suggested way that this might work, I certainly could implement it.

- *Baseball counter* - "This might be too simple to bother with, but people who are umpires (I'm mainly thinking about the many folks who ump for little league games) use a little hand held clicker to keep track of balls, strikes, and outs. This should be an easy applet to create." *David M. Schreck* <dschreck@csfbg.csfb.com> This is one where I would love to hear from someone who would actually use it. I have a number of ideas for user interface, but that would really depend on how someone would use it.

- *Tennis counter* - "Say I'm about to start a tennis game. I hit one button each time I score a point, and a different one each time my opponent scores. The applet always displays the current score. It might even display the word "deuce" when appropriate. Hopefully it could be programmed to be smart enough to know when subsequent games begin, and even keep track of the set score." *David M. Schreck* <dschreck@csfbg.csfb.com>. Here is where I will let my lack of knowledge of tennis show. I simply don't know how the scoring works well enough to write this. I would like to have the person enter the two names of the people playing and it would keep track of who has to serve, the current score, and the total match/set score. If someone would toss me this information, I could create the app really quickly.

- *Calorie Counter* - "If someone wanted to keep track of their caloric intake for the day (or any other need where you want to tally up a total but don't feel like carrying around a paper and pencil) perhaps they could just punch in the number to be added to the daily total each time they eat something. At the end of the day they can glance at the total and then reset to zero. *David M. Schreck* <dschreck@csfbg.csfb.com>" This is probably one of the more interesting apps to create. I might even take advantage of the EEProm to store some of the basic foods and their calorie counts to make it easier.

Wristapp Programming Tutorial

A First Wristapp - Hello World

To illustrate, let us take our favorite C Program and figure out how to put it on the Datalink. The first step in creating a wristapp is to decide on what the user interface will be. You would think that with only 5 buttons, this would be an easy task, but in reality this can make or break a good application. For our application, we will have it so that when you first enter the app, it puts "HELLO WORLD MODE" on the screen. If you press the PREV button, it will toggle to turning on all segments. Pressing the PREV button will switch back to the "HELLO WORLD MODE". The Next button will take you out of the app and the SET/NEXT buttons will not do anything. Pressing the GLOW button will activate the Indiglo light as expected. Here's what the code would look like:

```
;Name: Hello World
;Version: HELLO
;Description: This is a simple Hello Program
;by John A. Toebes, VIII
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 0 indicates that we want to show the segments instead of the message
;
START   EQU   *
;
; (2) System entry point vectors
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
        nop
        nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
        nop
        nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
        rts

L0123:  jmp     HANDLE_STATE0
        db      STATETAB-STATETAB
;
; (3) Program strings
S6_HELLO:   timex6  "HELLO "
```

```
S6_WORLD:   timex6  “WORLD “
;
; (4) State Table
; (4) State Table
STATETAB:
                db      0
                db      EVT_ENTER,TIM_ONCE,0    ; Initial state
                db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
                db      EVT_DNNEXT,TIM_ONCE,0   ; Next button
                db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
                db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.  We only see ENTER, RESUME, and DNNEXT events
;
HANDLE_STATE0:
                bset    1,$8f                   ; Indicate that we can be suspended
                lda     BTNSTATE                ; Get the event
                cmp     #EVT_DNNEXT             ; Did they press the next button?
                beq     DOTOGGLE                ; Yes, toggle what we are displaying
CLEARIT         bclr    0,FLAGBYTE              ; Start us in the show display state
REFRESH         brclr   0,FLAGBYTE,SHOWDISP     ; Do we want to see the main display?
                jmp     SETALL                  ; No, just turn on all segments
SHOWDISP        jsr     CLEARALL                ; Clear the display
                lda     #S6_HELLO-START         ; Get the offset for the first string
                jsr     PUT6TOP                 ; And send it to the top line
                lda     #S6_WORLD-START         ; Get the offset for the second string
                jsr     PUT6MID                 ; and put it on the middle line
                lda     #SYS8_MODE              ; Get the system offset for the ‘MODE’ string
                jmp     PUTMSGBOT               ; and put it on the bottom line
;
; (6) Our only real piece of working code...
DOTOGGLE        brset   0,FLAGBYTE,CLEARIT      ; If it is set, just jump to clear it like normal
                bset    0,FLAGBYTE              ; Already clear, so set it
                bra     REFRESH                 ; and let the refresh code handle it
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
                lda     #$c0                    ; We want button beeps and to indicate that we have been
loaded
                sta     $96
                clr     FLAGBYTE                ; start with a clean slate
                rts
```

Now all of that code needs a little explanation. As you can see from the numbers, we have 7 basic sections

1. Program specific constants - This is where you declare everything that you want to use. As a Wristapp, you have only a limited amount of Ram (7 bytes to be specific) that you can store your stuff with, so be careful here.

2. System entry point vectors - These are fixed and mandated for any Wristapp. If there is more than one state, the JMP and db sequence is repeated for each state.

3. Program strings - In order to provide addressability to the strings, you need to put them immediately after the entry point vectors.

4. State Table(s) - This really tells the watch how we want to operate and what events we want to handle. See **The State Table** for a more complete explanation of this.

5. State Table Handler(s) - These are called to process the events for a particular state. Typically this is a LDA BTNSTATE followed by a lot of CMP/Bcc instructions. You also need to do the BSET 1,$8f at the start to allow the Wristapp to be suspendable.

6. Program Specific Code - The actual meat of the program. In our case, we simply have to toggle a value.

7. Main Initialization routine - This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

Now that we have a basic program working. Next Up: **Getting Input - Numbers**

Getting Input

A program which just does output and really takes no input is not very useful. The first stage in making a program more useful is to figure out how to allow the user to enter a value. With this first numbers program, we allow you to enter a number by pressing the PREV/NEXT key to advance it by one each time you press the key. This allows us to see how basic input works and a couple of the formatting/display routines.

```
;Name: Numbers
;Version: NUMBER
;Description: This is a simple number count program
;by John A. Toebes, VIII
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE “WRISTAPP.I”
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 0 indicates that we want to show the segments instead of the message
;
CURVAL      EQU   $62   ; The current value we are displaying
START           EQU   *
;
; (2) System entry point vectors
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
        nop
        nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
        nop
        nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
        rts

L0123:  jmp HANDLE_STATE0
db  STATETAB-STATETAB
;
; (3) Program strings
S6_NUMBER:      timex6  “NUMBER”
S6_COUNT:   timex6  “COUNT “
;
; (4) State Table
```

```
STATETAB:
        db      0
        db      EVT_ENTER,TIM2_8TIC,0   ; Initial state
        db      EVT_TIMER2,TIM_ONCE,0    ; The timer from the enter event
        db      EVT_RESUME,TIM_ONCE,0    ; Resume from a nested app
        db      EVT_DNNEXT,TIM_ONCE,0   ; Next button
        db      EVT_DNPREV,TIM_ONCE,0    ; Prev button
        db      EVT_DNSET,TIM_ONCE,0     ; Set button
        db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.  We will see ENTER, RESUME, DNNEXT, DNPREV, DNSET, and
TIMER2
;
HANDLE_STATE0:
        bset    1,APP_FLAGS              ; Indicate that we can be suspended
        lda     BTNSTATE                 ; Get the event
        cmp     #EVT_DNNEXT              ; Did they press the next button?
        beq     DO_NEXT                 ; Yes, increment the counter
        cmp     #EVT_DNPREV              ; How about the PREV button
        beq     DO_PREV                  ; handle it
        cmp     #EVT_DNSET               ; Maybe the set button?
        beq     DO_SET                   ; Deal with it!
        cmp     #EVT_ENTER               ; Is this our initial entry?
        bne     REFRESH
;
; This is the initial event for starting us
;
DO_ENTER
        bclr    1,FLAGBYTE               ; Indicate that we need to clear the display
        jsr     CLEARSYM                         ; Clear the display
        lda     #S6_NUMBER-START
        jsr     PUT6TOP
        lda     #S6_COUNT-START
        jsr     PUT6MID
        lda     #SYS8_MODE
        jmp     PUTMSGBOT
;
; (6) Our only real working code...
DO_NEXT
        inc     CURVAL
        lda     CURVAL
        cmp     #100
        bne     SHOWVAL
DO_SET
clr     CURVAL
SHOWVAL
brset   1,FLAGBYTE,NOCLEAR
REFRESH
        jsr     CLEARALL
        bset    1,FLAGBYTE
NOCLEAR
        ldx     CURVAL
        jsr     FMTXLEAD0
        jmp     PUTMID34
DO_PREV
        lda     CURVAL
        beq     WRAPUP
        dec     CURVAL
        bra     SHOWVAL
WRAPUP
        lda     #99
        sta     CURVAL
        bra     SHOWVAL
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     FLAGBYTE                        ; start with a clean slate
        clr     CURVAL
rts
```

We have the same 7 basic sections, but some of them are a little more filled out.

1. Program specific constants - We have only two basic variables. The flagbyte and the current value.

2. System entry point vectors - We have nothing special this time..

3. Program strings - The strings go here for addressability.

4. State Table(s) - This really tells the watch how we want to operate and what events we want to handle. See **The State Table** for a more complete explanation of this. For this, we want to see the down events for the NEXT, PREV, and SET buttons so that we can increment, decrement, or reset the counter as appropriate. We also have coded the MODE button with the magic $FF which causes it to advance to the next app.

5. State Table Handler(s) - Here we have the typical CMP/BEQ instruction sequence to quickly determine what event happened. Note that the EVT_ENTER event causes a timer to go off which allows us to clear the screen 8/10 second after they switch to the app.

6. Program Specific Code - The actual meat of the program. We really only have to deal with advance/retreat/reset of the value and then displaying it after each change..

7. Main Initialization routine - This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

Just pressing a button for each increment can be tedious. Learn how to make it better with: **Better Input - Update**

Better Input - Update

Pressing the button for each time you want to increment or decrement a number can be very tedious. Fortunately, the Datalink has a series of update routines that you can call to handle this automatically. The update routine takes a few parameters. First is the type of update to do. The function limits

```
:
;Name: Update
;Version: UPDATE
;Description: This is a simple number update program
;by John A. Toebes, VIII
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 1 indicates that we need to clear the display first
;
CURVAL  EQU   $62       ; The current value we are displaying
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB-STATETAB
;
; (3) Program strings
S6_UPDATE:      timex6  "UPDATE"
S6_SAMPLE:      timex6  "SAMPLE"
;
```

```
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM_2_8TIC,0          ; Initial state
        db      EVT_TIMER2,TIM_ONCE,0           ; The timer from the enter event
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_DNANY4,TIM_ONCE,0           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,0           ; NEXT, PREV, SET, MODE button released
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, RESUME, DNANY4 and UPANY4 events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                      ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_DNANY4                     ; Did they press a button?
        bne     CHKENTER                        ; No, pass on to see what else there might be
        lda     BTN_PRESSED                     ; Let's see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_SET                        ; Perhaps the SET button
        beq     DO_SET                          ; If so, handle it
; In reality, we can't reach here since we handled all three buttons
; in the above code (the MODE button is handled before we get here and the
; GLOW button doesn't send in an event for this).  We can just fall through
; and take whatever we get from it.
CHKENTER
        cmp     #EVT_ENTER                      ; Is this our initial entry?
        bne     REFRESH
;
; This is the initial event for starting us
;
DO_ENTER
        bclr    1,FLAGBYTE                      ; Indicate that we need to clear the display
        jsr     CLEARSYM                        ; Clear the display
        lda     #S6_UPDATE-START
        jsr     PUT6TOP
        lda     #S6_SAMPLE-START
        jsr     PUT6MID
        lda     #SYS8_MODE
        jmp     PUTMSGBOT
;
; (6) Our real working code...
DO_NEXT
        bset    0,SYSFLAGS      ; Mark our update direction as up
        bra     DO_UPD
```

```
DO_PREV
bclr    0,SYSFLAGS      ; Mark our update direction as down
DO_UPD
clra
        sta     UPDATE_MIN      ; Our low end is 0
        lda     #99
        sta     UPDATE_MAX      ; and the high end is 99 (the max since this is a 2 digit value)
        ldx     #CURVAL         ; Point to our value to be updated
        lda     #UPD_MID34      ; Request updating in the middle of the display
        jsr     START_UPDATEP   ; And prepare the update routine
        bset    4,BTNFLAGS      ; Mark that the update is now pending
        bclr    1,FLAGBYTE
        lda     #SYS8_SET_MODE
        jmp     PUTMSGBOT

DO_SET
clr     CURVAL          ; When they hit the set button, we just clear to zero
SHOWVAL
brset   1,FLAGBYTE,NOCLEAR ; Do we need to clear the display first?
REFRESH
        jsr     CLEARALL        ; Yes, clear everything before we start
        bset    1,FLAGBYTE      ; And remember that we have already done that
NOCLEAR
        bclr    7,BTNFLAGS      ; Turn off any update routine that might be pending
        ldx     CURVAL          ; Get the current value
        jsr     FMTXLEAD0       ; Convert it to the two ASCII digits
        jmp     PUTMID34        ; And put it on the screen in the right place
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     FLAGBYTE                        ; start with a clean slate
        clr     CURVAL
rts
```

Now all of that code needs a little explanation. As you can see from the numbers, we have 7 basic sections

1. Program specific constants - This is where you declare everything that you want to use. As a Wristapp, you have only a limited amount of Ram (7 bytes to be specific) that you can store your stuff with, so be careful here.

2. System entry point vectors - These are fixed and mandated for any Wristapp. If there is more than one state, the JMP and db sequence is repeated for each state. We haven't started getting fancy so we still have only one state table.

3. Program strings - In order to provide addressability to the strings, you need to put them immediately after the entry point vectors. Our only strings are the two banner strings.

4. State Table(s) - This really tells the watch how we want to operate and what events we want to handle. See **The State Table** for a more complete explanation of this. We accept the normal RESUME, ENTER, and TIMER2 events for getting us running. We also handle the MODE button by allowing it to just bounce us out of the application and into the next. It is important that this event be in the table before the EVT_DNANY4 which allows for the NEXT, PREV, SET, and MODE buttons (it ignores the INDIGLO button). If you press the mode button, it will be handled by the first entry and the application terminated cleanly. Otherwise, we have to sort out which of the three buttons was pressed. This is easy to do since BTN_PRESSED holds the actual code associated with the button that was selected.

5. State Table Handler(s) - These are called to process the events for a particular state. Typically this is a LDA BTNSTATE followed by a lot of CMP/Bcc instructions. You also need to do the BSET 1,$8f at the start to allow the Wristapp to be suspendable. In this case we introduce the use of the EVT_DNANY4 in the basic state table logic testing. When we see the EVT_DNANY4 or EVT_UPANY4, we look at BTN_PRESSED to identify what the user pressed.

6. Program Specific Code - The actual meat of the program. Again, the code is very simple. We have to handle making sure that the screen is cleared at the appropriate times, but other than that, the majority of the work is picking a direction and setting 0.SYSFLAGS appropriately before letting the system handle the Update for us. Once we are set up, we set 4,BTNFLAGS and the system roms will handle updating the number for us.

7. Main Initialization routine - This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

This has gotten a bit better for input, now you need to show them what they have selected with: **Showing Selection - Blink**

Showing Selection - Blink routines

We can make our update program a bit smarter and more obvious to the user by blinking the digit when it is available to be changed. Like the START_UPDATEP routine, there is an equivalent START_BLINKP routine which handles blinking the display for you. I call this routine FLASH since it is not possible to put a K on the top two lines of the display :-).

```
;Name: Flash
;Version: FLASH
;Description: by John A. Toebes, VIII
;This is a simple number update/flash program
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 1 indicates that we need to clear the display first
;
CURVAL  EQU   $62       ; The current value we are displaying
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB-STATETAB
;
; (3) Program strings
S6_FLASH:       timex6  "FLASH "
S6_SAMPLE:      timex6  "SAMPLE"
;
```

```
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM_2_8TIC,0          ; Initial state
        db      EVT_TIMER2,TIM_ONCE,0           ; The timer from the enter event
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_DNANY4,TIM_ONCE,0           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,0           ; NEXT, PREV, SET, MODE button released
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, RESUME, DNANY4 and UPANY4 events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                       ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_DNANY4                     ; Did they press a button?
        bne     CHKENTER                        ; No, pass on to see what else there might be
        lda     BTN_PRESSED                     ; Let’s see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_SET                        ; Perhaps the SET button
        beq     DO_SET                          ; If so, handle it
; In reality, we can’t reach here since we handled all three buttons
; in the above code (the MODE button is handled before we get here and the
; GLOW button doesn’t send in an event for this).  We can just fall through
; and take whatever we get from it.
CHKENTER
        cmp     #EVT_ENTER                      ; Is this our initial entry?
        bne     REFRESH
;
; This is the initial event for starting us
;
DO_ENTER
        bclr    1,FLAGBYTE                      ; Indicate that we need to clear the display
        jsr     CLEARSYM                        ; Clear the display
        lda     #S6_FLASH-START
        jsr     PUT6TOP
        lda     #S6_SAMPLE-START
        jsr     PUT6MID
        lda     #SYS8_MODE
        jmp     PUTMSGBOT
;
; (6) Our real working code...
DO_NEXT
        bset    0,SYSFLAGS      ; Mark our update direction as up
        bra     DO_UPD
```

```
DO_PREV
bclr    0,SYSFLAGS      ; Mark our update direction as down
DO_UPD
clra
        sta     UPDATE_MIN      ; Our low end is 0
        lda     #99
        sta     UPDATE_MAX      ; and the high end is 99 (the max since this is a 2 digit value)
        ldx     #CURVAL         ; Point to our value to be updated
        lda     #UPD_MID34      ; Request updating in the middle of the display
        jsr     START_UPDATEP   ; And prepare the update routine
        bset    4,BTNFLAGS      ; Mark that the update is now pending
        bclr    1,FLAGBYTE
        lda     #SYS8_SET_MODE
        jmp     PUTMSGBOT

DO_SET
clr     CURVAL          ; When they hit the set button, we just clear to zero
SHOWVAL
brset   1,FLAGBYTE,NOCLEAR ; Do we need to clear the display first?
REFRESH
        jsr     CLEARALL        ; Yes, clear everything before we start
        bset    1,FLAGBYTE      ; And remember that we have already done that
NOCLEAR
        bclr    7,BTNFLAGS      ; Turn off any update routine that might be pending
        ldx     #CURVAL
        lda     #BLINK_MID34
        jsr     START_BLINKP
        bset    2,BTNFLAGS      ; Mark a blink routine as pending
rts
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     FLAGBYTE                        ; start with a clean slate
        clr     CURVAL
rts
```

This is code is basically identical to the **Update** sample with only a couple of minor changes.

1. Program specific constants - No Change.

2. System entry point vectors - We have nothing special this time..

3. Program strings - Gee, we changed the strings.

4. State Table(s) - We get to use exactly the same state table. See **The State Table** for a more complete explanation of this.

5. State Table Handler(s) - Since the state table is the same, the state handling is the same.

6. Program Specific Code - All we had to do different here was to call START_BLINKP and then set 2,BTNFLAGS to notify the system that we want the blink routine to run. The blink routine will automatically handle putting up the number for us.

7. Main Initialization routine - No changes here either. This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

Entering Digits - PASSWD sample

This program is a bit more sophisticated to show off how you might go toward creating a complex app. I have not made any attempts at optimizing the code here in order to be a bit more clear about how to go about writing this type of app. There are a few new features with this code:

- We have two different display screens. When you first enter the app, it puts up one display. After it times out, it puts up a different display which also has a scrolling message across the bottom.
- The set button brings you into a set mode where the mode button switches between digits to set.
- This app uses two state tables instead of one. It shows how to switch between the two states.

```
;Name: Password
;Version: PASSWD
;Description: This is a simple number update/passwd program
;by John A. Toebes, VIII
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE “WRISTAPP.I”
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 0 indicates which digit we are working on (SET=SECOND DIGIT)
;   Bit 1 indicates that we need to clear the display first
;
DIGIT0  EQU     $62     ; The first digit to enter
DIGIT1  EQU     $63     ; The second digit to enter
SYSTEMP0        EQU     $A0
SYSTEMP1        EQU     $A1
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop
```

```
L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB0-STATETAB0
L0127:  jmp     HANDLE_STATE1
        db      STATETAB1-STATETAB0
;
; (3) Program strings
S6_TOEBES:      timex6  “TOEBES”
S6_SAMPLE:      timex6  “SAMPLE”
S6_PRESS:       timex6  “PRESS “
S8_PASSWORD:    Timex   “PASSWORD”
SX_MESSAGE      Timex   “BY JOHN A. TOEBES, VIII”
                db      SEPARATOR
;
; (4) State Table
;
STATETAB0:
        db      0
        db      EVT_ENTER,TIM_2_8TIC,0          ; Initial state
        db      EVT_TIMER2,TIM_ONCE,0           ; The timer from the enter event
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_SET,TIM_ONCE,1              ; SET button pressed
        db      EVT_END

STATETAB1:
        db      1
        db      EVT_RESUME,TIM_ONCE,1           ; Resume from a nested app
        db      EVT_DNANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button released
        db      EVT_USER2,TIM_ONCE,0
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_ENTER                      ; Is this our initial entry?
        bne     REFRESH0
;
; This is the initial event for starting us
;
DO_ENTER
        bclr    1,FLAGBYTE                      ; Indicate that we need to clear the display
        jsr     CLEARSYM                        ; Clear the display
        lda     #S6_TOEBES-START
        jsr     PUT6TOP
        lda     #S6_SAMPLE-START
        jsr     PUT6MID
        lda     #S8_PASSWORD
        jmp     BANNER8
;
; We come here for a RESUME or TIMER2 event.  For this we want to reset the display
;
REFRESH0
        brset   1,FLAGBYTE,NOCLEAR0             ; Do we need to clear the display first?
        bset    1,FLAGBYTE
        jsr     CLEARSYM
NOCLEAR0
        lda     #S6_PRESS-START
        jsr     PUT6TOP
        lda     #SYS6_SET
        jsr     PUTMSG2
        lda     #SX_MESSAGE-START
        jmp     SETUP_SCROLL
;
; (6) State Table 1 Handler
; This is called to process the state events.
; We see SET, RESUME, DNANY4, and UPANY4 events
;
HANDLE_STATE1:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_UPANY4
        beq     REFRESH
        cmp     #EVT_DNANY4                     ; Is this our initial entry?
        bne     FORCEFRESH
        lda     BTN_PRESSED                     ; Let's see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_MODE                       ; Perhaps the MODE button
        beq     DO_MODE                         ; If so, handle it
; It must be the set button, so take us out of this state
        lda     #EVT_USER2
        jmp     POSTEVENT
;
; (7) Our real working code...
DO_NEXT
        bset    0,SYSFLAGS                      ; Mark our update direction as up
        bra     DO_UPD
DO_PREV
bclr    0,SYSFLAGS                      ; Mark our update direction as down
DO_UPD
clra
        sta     UPDATE_MIN                      ; Our low end is 0
        lda     #99
        sta     UPDATE_MAX                      ; and the high end is 99 (the max since this is a 2 digit
value)
        brset   0,FLAGBYTE,UPD1
        ldx     DIGIT1
        jsr     FMTXLEAD0
        jsr     PUTMID34
        ldx     #DIGIT0                         ; Point to our value to be updated
        lda     #UPD_MID12                      ; Request updating in the middle of the display
        bra     UPD2
UPD1
        ldx     DIGIT0
        jsr     FMTXLEAD0
        jsr     PUTMID12
        ldx     #DIGIT1
        lda     #UPD_MID34
UPD2
        jsr     START_UPDATEP   ; And prepare the update routine
        bset    4,BTNFLAGS      ; Mark that the update is now pending
        bclr    1,FLAGBYTE
        lda     #SYS8_SET_MODE
        jmp     PUTMSGBOT

DO_MODE
        lda     FLAGBYTE
        eor     #1
        sta     FLAGBYTE

REFRESH
brset   1,FLAGBYTE,NOCLEAR ; Do we need to clear the display first?
FORCEFRESH
        jsr     CLEARALL        ; Yes, clear everything before we start
        bset    1,FLAGBYTE      ; And remember that we have already done that
NOCLEAR
        bclr    7,BTNFLAGS      ; Turn off any update routine that might be pending
        brset   0,FLAGBYTE,SET1
        ldx     DIGIT1
        jsr     FMTXLEAD0
        jsr     PUTMID34
        ldx     #DIGIT0
        lda     #BLINK_MID12
        bra     SET2
SET1
        ldx     DIGIT0
        jsr     FMTXLEAD0
        jsr     PUTMID12
        ldx     #DIGIT1
        lda     #BLINK_MID34
SET2
        jsr     START_BLINKP
        bset    2,BTNFLAGS      ; Mark a blink routine as pending
rts
;
```

```
; (8) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     FLAGBYTE                        ; start with a clean slate
        clr     DIGIT0
        clr     DIGIT1
rts
;
; (9) This subroutine is useful for getting a scrolling string on the screen
;
;----------------------------------------------------------------------
; Routine:
;   SETUP_SCROLL
; Parameters:
;   X - Offset from Start to the string
; Returns:
;   MSGBUF - contains copied string
; Purpose
;   This copies the current string into MSGBUF and calls the appropriate routines
;   to start it scrolling on the bottom line.
;----------------------------------------------------------------------
SETUP_SCROLL:
        clr     SYSTEMP0
        sta     SYSTEMP1
DO_COPY:
        ldx     SYSTEMP1        ; Get the pointer to the source character
        lda     START,X         ; Get the character that we are copying
        ldx     SYSTEMP0        ; Get the pointer to the output buffer
        sta     MSGBUF,X        ; and store the character away
        inc     SYSTEMP0        ; Increment our count
        inc     SYSTEMP1        ; As well as the pointer to the character
        cmp     #SEPARATOR      ; Did we get a terminator character
        bne     DO_COPY         ; No, go back for more
        ;
; The string is now in a buffer terminated by a separator character
        ;
        jsr     PUTSCROLLMSG        ; Initialize the scrolling support
        jmp     SCROLLMSG           ; And tell it to actually start scrolling
```

This is code is built on the Update and Blink samples with a few changes and additions.

1. Program specific constants - We now have two digits to care about.

2. System entry point vectors - Because we have gone to two state tables, we now have the extra jump vector.

3. Program strings - Gee, we changed the strings. Plus we have a longer string which we pass to our SETUP_SCROLL routine.

4. State Table(s) - We now have two state tables. State table0 is pretty simple and is used only for when we are in the normal state. State table 1 is used when we are in the set mode. See **The State Table** for a more complete explanation of this.

5. State Table Handler0 - For state0, we only really need to handle the initial enter where we put up the banner. After a while we time out and put up the 'PRESS SET' message with my name scrolling across the bottom.

6. State Table Handler1 - This handler is used for when we are in the SET state for changing the numbers.

7. Program Specific Code - We use the same UPDATE and BLINK functions from the Blink sample. The only extra work here is that we cause the display to update the other digit when we are setting one.

8. Main Initialization routine - No changes here. This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

9. SETUP_SCROLL subroutine - This is a useful routine that you may wish to copy for another wristapp.

Getting time and Input - DAYFIND sample

This is the first real app with some attempt at optimization and a bit of planning for user input. It stems from a suggestion by Roman Mazi. There are a lot of things in this code which build on the previous examples. The most notable things in this one are:

- This code shows how to get the current date (and you can also get the time the same way).
- There are banner messages on the bottom of the display to provide a little help.
- Workarounds for a lack of update routines are given.
- Quite a few new routines are introduced here.

The code is reasonably commented:

```
;Name: Day Finder
;Version: DAYFIND
;Description: This will allow you to determine the date for a given day of the week and vice-versa.
;by John A. Toebes, VIII
;
;Press the prev/next buttons to advance by a single day. Press SET to access the ability to advance/backup
by
;weeks, months, days, and years.  The MODE button advances through those different states
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
B_CLEAR         EQU     0       ; Bit 0 indicates that we need to clear the display first
B_SCANUP        EQU     1       ; Bit 1 indicates that we are scanning up
B_SCANNING      EQU     2       ; Bit 2 indicates that we are in a fake scanning mode
DIGSEL          EQU     $62     ; Indicates which digit we are working on
                                ; 0 = DAY OF WEEK
                                ; 1 = Month
                                ; 2 = Day
                                ; 3 = Year
YEAR_DIG1       EQU     $63     ; This is the first digit of the year to blink (the tens digit)
YEAR_DIG2       EQU     $64     ; This is the second digit of the year to blink (the ones digit)
COUNTER         EQU     $65     ; A convenient counter for us to advance a week at a time
;
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
```

```
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB0-STATETAB0
L0127:  jmp     HANDLE_STATE1
        db      STATETAB1-STATETAB0
;
; (3) Program strings
S6_DAY          timex6  “DAY “
S6_FIND         timex6  “  FIND”
S8_TOEBES       Timex   “J.TOEBES”
S8_DAYFIND      Timex   “DAY FIND”
S8_WEEK         db      C_LEFTARR
                Timex   “ WEEK “
                db      C_RIGHTARR
S8_MONTH        db      C_LEFTARR
                Timex   “MONTH “
                db      C_RIGHTARR
S8_DAY          db      C_LEFTARR
                Timex   “ DAY  “
                db      C_RIGHTARR
S8_YEAR         db      C_LEFTARR
                Timex   “ YEAR “
                db      C_RIGHTARR
;
; (4) State Table
;
STATETAB0:
        db      0
        db      EVT_ENTER,TIM1_4TIC,0           ; Initial state
        db      EVT_TIMER1,TIM_ONCE,0           ; The timer from the enter event
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_SET,TIM_ONCE,1              ; SET button pressed
        db      EVT_DNNEXT,TIM2_8TIC,0          ; NEXT button pressed
        db      EVT_DNPREV,TIM2_8TIC,0          ; PREV button pressed
        db      EVT_UPANY4,TIM_ONCE,0           ; The
        db      EVT_TIMER2,TIM2_TIC,0           ; The timer for the next/prev button pressed
        db      EVT_END
```

```
STATETAB1:
        db      1
        db      EVT_RESUME,TIM_ONCE,1           ; Resume from a nested app
        db      EVT_DNANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button released
        db      EVT_USER2,TIM_ONCE,0
        db      EVT_USER3,TIM2_8TIC,1           ;
        db      EVT_TIMER2,TIM2_TIC,1           ;
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_DNNEXT
        beq     DO_NEXT0
        cmp     #EVT_DNPREV
        beq     DO_PREV0
        cmp     #EVT_TIMER2
        beq     DO_SCAN
        cmp     #EVT_ENTER                      ; Is this our initial entry?
        bne     REFRESH0
;
; This is the initial event for starting us up
;
DO_ENTER
;
; (6) This code gets the current date from the system
        jsr     ACQUIRE                         ; Lock so that it doesn't change under us
        ldx     #TZ1_MONTH                      ; Assume that we are using the first timezone
        jsr     CHECK_TZ                        ; See which one we are really using
        bcc     COPY_TZ1                        ; If we were right, just skip on to do the work
        ldx     #TZ2_MONTH                      ; Wrong guess, just load up the second time zone
COPY_TZ1
        lda     0,x                             ; Copy out the month
        sta     SCAN_MONTH
        lda     1,x                             ; Day
        sta     SCAN_DAY
        lda     2,x                             ; and year
        sta     SCAN_YEAR
        jsr     RELEASE                         ; Unlock so the rest of the system is happy

        bclr    B_CLEAR,FLAGBYTE                ; Indicate that we need to clear the display
        clr     DIGSEL                          ; Start us off on the week advance
        jsr     CLEARSYM                        ; Clear the display
        lda     #S6_DAY-START
        jsr     PUT6TOP
        lda     #S6_FIND-START
        jsr     PUT6MID
        lda     #S8_TOEBES-START
        jmp     BANNER8

DO_SCAN
brclr   B_SCANUP,FLAGBYTE,DO_PREV0      ; Were we scanning up or down?
DO_NEXT0
        bset    B_SCANUP,FLAGBYTE               ; We are now scanning up
        jsr     INCREMENT_SCAN_DATE             ; Advance to the next date
        bra     SHOW_DATE                       ; Comment this out and use the next one if you want
        ;  jmp     APPT_SHOW_SCAN               ; to put the text ‘SCAN’ on the bottom when we are in scan
mode

DO_PREV0
        bclr    B_SCANUP,FLAGBYTE               ; We are now scanning down
        jsr     DECREMENT_SCAN_DATE             ; Back up to the previous date
        bra     SHOW_DATE                       ; Show the date on the screen.
        ;  jmp     APPT_SHOW_SCAN               ; Use this if you want ‘SCAN’ on the bottom of the display
;
; We come here for a RESUME or TIMER2 event.  For this we want to reset the display
;
REFRESH0
        brset   B_CLEAR,FLAGBYTE,NOCLEAR0       ; Do we need to clear the display first?
        bset    B_CLEAR,FLAGBYTE                ; Mark that the display has been cleared
        jsr     CLEARALL                        ; and do the work of clearing
NOCLEAR0
        lda     #S8_DAYFIND-START               ; Put up the name of the app on the display
        jsr     BANNER8
SHOW_DATE
        jsr     APPT_SHOW_DATE                  ; Show the date on the screen
        ldx     SCAN_YEAR                       ; as well as the year
        jmp     PUTYEARMID
;--------------------------------------------------------------------------------
; (7) State Table 1 Handler
; This is called to process the state events.
; We see SET, RESUME, USER3, TIMER2, DNANY4, and UPANY4 events
;  We use the USER3 to trigger a delay which fires off a TIMER2 sequence of events.
;  This allows us to have the PREV/NEXT buttons repeat for advancing the WEEK and YEAR
;  since we can’t use the UPDATE routines for them.
;
HANDLE_STATE1:
        bset    1,APP_FLAGS                    ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_TIMER2                     ; Was it a timer for a repeat operation?
        beq     DO_UPD                          ; Yes, go handle it
        cmp     #EVT_USER3                      ; Was it the USER3 event fired from the PREV/NEXT buttons?
        bne     TRY_UP                          ; No, try again
        rts                                     ; Yes, just ignore it, it will cause a timer to go off
later
TRY_UP
        bclr    B_SCANNING,FLAGBYTE             ; We can’t be scanning any more, so turn it off
        cmp     #EVT_UPANY4                     ; Was it any button being released?
        bne     TRY_DN                          ; No, try again
        jmp     REFRESH                         ; Yes, go refresh the screen (note that the branch is out
of range)
TRY_DN
        cmp     #EVT_DNANY4                     ; Is this our initial entry?
        beq     GET_DN                          ; No, try again
        jmp     FORCEFRESH                      ; Yes, go setup the screen (note that the branch is out of
range)
GET_DN
        lda     BTN_PRESSED                     ; Let's see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_MODE                       ; Perhaps the MODE button
        beq     DO_MODE                         ; If so, handle it
; It must be the set button, so take us out of this state
        lda     #EVT_USER2
        jmp     POSTEVENT
;
; (8) Our real working code...
; We come here when they press the next/prev buttons.  if we are in a timer repeat
; situation (triggered when they press prev/next for the WEEK/YEAR) then we skip right
; to processing based on the button that was previously pressed
;
DO_NEXT
        bset    0,SYSFLAGS                      ; Mark our update direction as up
        bra     DO_UPD
DO_PREV
bclr    0,SYSFLAGS                      ; Mark our update direction as down
DO_UPD
        lda     DIGSEL                          ; Which digit mode are we in?
        beq     DO_UPD_DOW                      ; 0 - Handle the WEEK
        cmp     #2
        blo     DO_UPD_MONTH                    ; <2 = 1 - Handle the MONTH
        beq     DO_UPD_DAY                      ; 2 - Handle the Day
DO_UPD_YEAR                                     ; >2 = 3 - Handle the YEAR
        brclr   0,SYSFLAGS,LASTYEAR             ; Were we in the down direction?
        ldx     #99                             ; Going up, let the WRAPX routine handle it for us
        lda     SCAN_YEAR
        jsr     INCA_WRAPX
        bra     SAVEYEAR
LASTYEAR
        lda     SCAN_YEAR                       ; Going down, get the year
        deca                                    ; Decrement it
        bpl     SAVEYEAR                        ; and see if we hit the lower end
        lda     #99                             ; Yes, 2000 wraps down to 1999
SAVEYEAR
        sta     SCAN_YEAR                       ; Save away the new year
        bra     SETUP_LAG                       ; And fire off an event to allow for repeating
```

```
DO_UPD_DOW                                      ; 0 - Day of week
        lda     #7                              ; We want to iterate 7 times advancing by one day.
        sta     COUNTER                         ;  (this makes it much easier to handle all the fringe
cases)
WEEKLOOP
        brclr   0,SYSFLAGS,LASTWEEK             ; Are we going backwards?
        jsr     INCREMENT_SCAN_DATE             ; Going forwards, advance by one day
        bra     WEEKLOOPCHK                     ; And continue the loop
LASTWEEK
jsr     DECREMENT_SCAN_DATE             ; Going backwards, retreat by one day
WEEKLOOPCHK
        dec     COUNTER                         ; Count down
        tst     COUNTER                         ; See if we hit the limit
        bne     WEEKLOOP                        ; and go back for more
; (9) Fake repeater
; This code is used for the Day of week and year modes where we want to have a
; repeating button, but the system routines won’t handle it for us
; It works by posting a USER3 event which has a timer of about ½ second.
; After that timer expires, we get a timer2 event which then repeats every tic.
; The only thing that we have to worry about here is to not go through this
; every time so that it takes ½ second for every repeat.
SETUP_LAG
        brset   B_SCANNING,FLAGBYTE,INLAG       ; If we were already scanning, skip out
        bset    B_SCANNING,FLAGBYTE             ; Indicate that we are scanning
        lda     #EVT_USER3                      ; and post the event to start it off
        jsr     POSTEVENT
INLAG
jmp     SHOW_DATE                       ; Put the date up on the display
; (10) Update routine usage
DO_UPD_MONTH                                    ; 1 - Handle the month
        lda     #MONTH_JAN                      ; The bottom end is January
        sta     UPDATE_MIN
        lda     #MONTH_DEC                      ; and the top end is December (INCLUSIVE)
        sta     UPDATE_MAX
        lda     #UPD_HMONTH                     ; We want the HALF-MONTH udpate function
        ldx     #SCAN_MONTH                     ; To update the SCAN_MONTH variable
        bra     SEL_UPD                         ; Go do it
DO_UPD_DAY                                      ; 2 - Handle the day
        lda     #1                              ; 1 is the first day of the month
        sta     UPDATE_MIN
        jsr     GET_SCAN_MONTHLEN               ; Figure out how long the month is
        sta     UPDATE_MAX                      ; and make that the limit
        lda     #UPD_HDAY                       ; We want the HALF-DAY update function
        ldx     #SCAN_DAY                       ; to update the SCAN_DAY variable
SEL_UPD
        jsr     START_UPDATEP                   ; And prepare the update routine
        bset    4,BTNFLAGS                      ; Mark that the update is now pending
rts
; (11) Making the mode button work
; when they press the mode button, we want to cycle through the various choices
; on the display.
```

```
DO_MODE
        lda     DIGSEL                          ; Figure out where we are in the cycle
        inca                                    ; advance to the next one
        and     #3                              ; and wrap at 4 to zero
        sta     DIGSEL
REFRESH
brset   B_CLEAR,FLAGBYTE,NOCLEAR        ; Do we need to clear the display first?
FORCEFRESH
        jsr     CLEARALL                        ; Yes, clear everything before we start
        bset    B_CLEAR,FLAGBYTE                ; And remember that we have already done that
NOCLEAR
        clr     BTNFLAGS                        ; Turn off any scrolling banners
        lda     #ROW_TD23                       ; Turn off the dash from the week blink
        sta     DISP_ROW
        bclr    COL_TD23,DISP_COL
        jsr     SHOW_DATE                       ; Display the date
; (12) Establishing a blink routine
; This makes the appropriate section of the display blink based on what we are changing
        lda     DIGSEL                          ; Get the digit we are on
        beq     DO_BLINK_DOW                    ; 0 -> Update Day of week
        cmp     #2
        blo     DO_BLINK_MONTH                  ; <2 = 1 -> Update month
        beq     DO_BLINK_DAY                    ; 2 - Update day of month

DO_BLINK_YEAR   ;        3: Year
; (13) Calling BLINK_SECOND
; For BLINK_SECONDS, the UPDATE_PARM points to the 2 character format for the year.
        ldx     SCAN_YEAR                       ; Get our year
        jsr     GETBCDHI                        ; And extract out the high digit of it
        sta     YEAR_DIG1                       ; Save that away
        ldx     SCAN_YEAR                       ; Do it again
        jsr     GETBCDLOW                       ; to get the low digit
        sta     YEAR_DIG2                       ; and save that away
        ldx     #YEAR_DIG1                      ; the parm points to the first digit
        lda     #BLINK_SECONDS                  ; and we want a BLINK_SECONDS function
        bra     SETUP_BLINK                     ; so do it already

DO_BLINK_DOW    ;        0: Day of week:
; (14) Calling BLINK_SEGMENT
; Unfortunately, there is no blink routine to blink the upper two letters on the display.
; To get around this, I have chosen to blink a single segment on the display (the dash
; after the day of the week).  This routine was designed to blink the AM/PM or other
; symbols, but it works quite fine for our purposed.  You need to set UPDATE_POS to have
; the row to be updated and UPDATE_VAL holds the mask for the COLUMS to be XORed.
; In this way, you might have more than one segment blinking, but there are few segments
; on the same row which would achieve a reasonable effect.
;            UPDATE_POS   ROW_TD23
;            UPDATE_VAL   (1<<COL_TD23)
        lda     #ROW_TD23
; We want to blink the DASH after the day of week sta UPDATE_POS
; Store the ROW for it in UPDATE_POS lda #(1<<COL_TD23)
; Get the mask for the column sta UPDATE_VAL
; And store that in UPDATE_VAL lda #BLINK_SEGMENT
; We want a BLINK_SEGMENT function bra SETUP_BLINK
; and get to it.
DO_BLINK_MONTH         ; 1: Month
; (15) Calling BLINK_HMONTH, BLINK_HDAY
; These are the normal boring cases of calling the blink routine.  They simply need the
; address of the byte holding the value to blink and the function to blink them with.
;             UPDATE_PARM - Points to the month
        lda     #BLINK_HMONTH                   ; We want a BLINK HALF-MONTH function
        ldx     #SCAN_MONTH                     ; to blink our month
        bra     SETUP_BLINK                     ; and do it

DO_BLINK_DAY    ;       2: Day
;            UPDATE_PARM - Points to the day
        lda     #BLINK_HDAY                     ; We want a BLINK HALF-DAY function
        ldx     #SCAN_DAY                       ; to blink our day

SETUP_BLINK
        jsr     START_BLINKP                    ; Request the blink function
        lda     digsel                          ; Figure out which one we are blinking
        lsla                                    ; *2
        lsla                                    ; *4
        lsla                                    ; *8
        add     #S8_WEEK-START                  ; And use that to index the banner to put on the bottom
        jsr     BANNER8
        bset    2,BTNFLAGS                      ; Mark a blink routine as pending
rts
;
; (16) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                           ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     FLAGBYTE                        ; start with a clean slate
rts
```

This is code is built on the passwd with a quite a few changes and additions.

1. Program specific constants - different uses for the flags and a couple of new local variables

2. System entry point vectors - No change here.

3. Program strings - Gee, we changed the strings. Note the four strings in a row which serve as help messages when in set mode.

4. State Table(s) - State table0 is not radically changed (We added the next/prev buttons). State table 1 is used when we are in the set mode. See **The State Table** for a more complete explanation of this. Note the use of the USER3 event in this table

5. State Table Handler0 - For state0, we only really need to handle the initial enter where we put up the banner. After a while we time out and put up the current day of the week and our banner.

6. Get the system date - This shows how to get the current date.
7. State table 1 handler
8. Program Specific Code - We use the same UPDATE and BLINK functions from the Blink sample.
9. Fake Repeater - I'm pretty proud of this one...
10. Update routine usage - Look here for some clues on using the update routines.
11. Making the mode button work
12. Establishing a blink routine
13. Calling BLINK_SECOND
14. Calling BLINK_SEGMENT
15. Calling BLINK_HMONTH, BLINK_HDAY
16. Main initialization - Surprisingly, there is not much change here.

Playing With Sound - TestSnd example

This is a very simple program that I had put together to test out what sounds the watch can make. The program doesn't really do a lot except poke the hardware a little. It does use the update routine without the blinking. Unlike programs which play a tune, this goes straight to the hardware to test out the capabilities and is completely independent of any sound scheme that you might have loaded.

```
;Name: Test Sound
;Version: TESTSND
;Description: This routine tests the various sound capabilities of the DataLink.
;by John A. Toebes, VIII
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
CURRENT_VAL        EQU         $61
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     DOEVENT0
        db      TABLE0-TABLE0
L0127:  jmp     DOEVENT1
        db      TABLE1-TABLE0
;
; (3) Program strings
S6_SOUND:       timex6  "SOUND "
S6_TEST:        timex6  " TEST "
S8_TOEBES:      Timex   "J.TOEBES"
;
; (4) State Table
```

```
;
TABLE0:
                db      0
                db      EVT_ENTER,TIM_LONG,0    ; Initial state
                db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
                db      EVT_TIMER2,TIM_ONCE,0   ;
                db      EVT_DNNEXT,TIM_ONCE,1   ; Next button
                db      EVT_DNPREV,TIM_ONCE,1   ; Prev button
                db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
                db      EVT_DNSET,TIM_ONCE,0    ; Set button
                db      EVT_UPSET,TIM_ONCE,0    ;
                db      EVT_END

TABLE1:
                db      1
                db      EVT_UPNEXT,TIM_ONCE,1   ; Releasing the next button
                db      EVT_UPPREV,TIM_ONCE,1   ; Releasing the prev button
                db      EVT_USER0,TIM_ONCE,0    ; Return to the main state table
                db      EVT_END                 ; End of table
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, and RESUME events
;
DOEVENT0:
                bset    1,APP_FLAGS             ; Allow us to be suspended
                lda     BTNSTATE                ; Get the event
                cmp     #EVT_RESUME             ; Did another app get called in the meantime?
                beq     REFRESH                 ; We will refresh the display in this case
                cmp     #EVT_TIMER2             ; Did the initial timer expire?
                beq     REFRESH                 ; Yes, clean up the screen
                cmp     #EVT_ENTER              ; Is this the initial state?
                beq     INITBANNER              ; Yes, put up the banner
                cmp     #EVT_DNSET              ; Did they hit the set button
                beq     PLAYIT
                cmp     #EVT_UPSET
                beq     SILENCE
rts
;
; (6) Sound playing code.  Note that we go straight to the hardware here for this one
;
PLAYIT:
                lda     #ROW_NOTE               ; Turn on the little note symbol
                sta     DISP_ROW
                bset    COL_NOTE,DISP_COL
                lda     CURRENT_VAL
                sta     $28
rts
SILENCE:
                lda     #ROW_NOTE               ; Turn off the little note symbol
                sta     DISP_ROW
                bclr    COL_NOTE,DISP_COL
                lda     #15
                sta     $28
rts
REFRESH:
                jsr     CLEARALL                ; Clear the display
                lda     #S6_SOUND-START         ; Put “SOUND” on the top of the display
                jsr     PUT6TOP
                ldx     CURRENT_VAL
                jsr     FMTX
                jsr     PUTMID34
                bra     JBANNER

INITBANNER:
                jsr     CLEARALL                ; Clear the display
                lda     #S6_SOUND-START         ; Put ‘SOUND ‘ on the top line
                jsr     PUT6TOP
                lda     #S6_TEST-START          ; Put ‘ TEST ‘ on the second line
                jsr     PUT6MID
JBANNER
                lda     #S8_TOEBES-START
                jmp     BANNER8
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
                bset    7,WRISTAPP_FLAGS        ; Tell them that we are a live application
                clr     CURRENT_VAL
rts
;
; (8) State Table 1 Handler
;
; This is called when we press the prev/next button or when the timer fires during that event
;
DOEVENT1:
                lda     BTNSTATE
                cmp     #EVT_DNPREV
                beq     GO_DOWN
                cmp     #EVT_DNNEXT
                beq     GO_UP
                lda     #EVT_USER0
                jmp     POSTEVENT

GO_DOWN         bclr    0,SYSFLAGS      ; Mark update direction as down
                bra     DOUPDN
GO_UP           bset    0,SYSFLAGS      ; Mark update direction as up
DOUPDN          clra
                jsr     CLEARMID
                sta     UPDATE_MIN
                lda     #99
                sta     UPDATE_MAX
                ldx     #CURRENT_VAL
                lda     #UPD_MID34
                jsr     START_UPDATEP
                bset    4,BTNFLAGS
rts
```

This code has a few notable sections.

1. Program specific constants - Nothing special here

2. System entry point vectors - Nothing new here either.

3. Program strings - Of course we changed the strings once again.

4. State Table(s) - We have two state tables. Both of these are pretty simple. StateTable0 has a lot of values instead of using the EVT_DNANY event just for a little variety. StateTable1 is used just for the increment/decrement mode. See **The State Table** for a more complete explanation of this.

5. State Table Handler0 - For state0, we only really need to handle the initial enter where we put up the banner. It times out and puts up the sound banner. When you press the set button, it will play the sound.

6. Sound playing code - This code simply pokes the current value to the hardware at $28. When we let go of the button, we make the hardware silent by poking a $0f to that same location.

7. Main Initialization routine - Nothing really significant here. This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS.

8. State Table Handler1 - Nothing really significant here, it uses the same update routines that most of the other examples use.

Using Callbacks - Endoff example

Here is another pretty simple program that shows off a couple of useful features of a wristapp. This one stems from a request several people have had (including myself) to turn off the alarms on the weekend. That's really all this does. To make it a little more fun, I decided that I wanted to call it " WEEK " "ENDOFF", with the problem that there is no letter K in the character set for the top line on the display. So, I figured out how to make a reasonably ok looking letter. You will notice that this program seems to do very little...

```
;Name: Week End Off
;Version: ENDOFF
;Description: Week End Off - by John A. Toebes, VIII
;This application turns off all alarms on the weekend.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
START     EQU   *
;
; (2) System entry point vectors
;
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  jmp     CHECKSTATE      ; Called to handle any timers or time events - WRIST_DOTIC
L0119:  jmp     ENABLE_ALL      ; Called when the COMM app starts and we have timers pending -
WRIST_INCOMM
L011c:  jmp     CHECKSTATE      ; Called when the COMM app loads new data - WRIST_NEWDATA

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB-STATETAB
;
; (3) Program strings
;
S6_WEEK:        timex6  " WEEH "
S6_ENDOFF:      timex6  "ENDOFF"
S8_TOEBES:      Timex   "J.TOEBES"
;
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM_LONG,0    ; Initial state
        db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS             ; Allow us to be suspended
        jsr     CLEARALL                ; Clear the display
        lda     #S6_WEEK-START          ; Put ‘ WEEK ‘ on the top line
        jsr     PUT6TOP
        lda     #S6_ENDOFF-START        ; Put ‘ENDOFF’ on the second line
        jsr     PUT6MID
;
; (6) Faking a letter K
;
;
; We have    We want it to look like:
; |     |    |
; |     |    |  |
; |     |    |  |
; |=====|    |=====
; |     |    |     |
; |     |    |     |
; |     |    |     |
; This means turning off T5B and turning on T5H
        lda     #ROW_T5B
        sta     DISP_ROW
        bclr    COL_T5B,DISP_COL
        lda     #ROW_T5H
        sta     DISP_ROW
        bset    COL_T5H,DISP_COL
        jsr     CHECKSTATE              ; Just for fun, check the alarm state
        lda     #S8_TOEBES-START
        jmp     BANNER8
;
; (7) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        bset    7,WRISTAPP_FLAGS        ; Tell them that we are a live application
        lda     #$C8    ; Bit3 = wristapp wants a call once a day when it changes (WRIST_DOTIC) (SET=CALL)
                        ; Bit6 = Uses system rules for button beep decisions (SET=SYSTEM RULES)
                        ; Bit7 = Wristapp has been loaded (SET=LOADED)
        sta     WRISTAPP_FLAGS
; Fall into CHECKSTATE
;
; (8) Determining the day of the week
;
CHECKSTATE
        jsr     ACQUIRE                 ; Lock so that it doesn’t change under us
        lda     TZ1_DOW                 ; Assume that we are using the first timezone
        jsr     CHECK_TZ                ; See which one we are really using
        bcc     GOT_TZ1                 ; If we were right, just skip on to do the work
        lda     TZ2_DOW                 ; Wrong guess, just load up the second time zone
GOT_TZ1
        jsr     RELEASE                 ; Unlock so the rest of the system is happy
        cmp     #5                      ; Time zone day of week is 0=Monday...6=Sunday
        bhs     DISABLE_ALL             ; Saturday, Sunday - disable them all
; Fall into ENABLE_ALL
;---------------------------------------------------------------
; Routine:
;   (9) ENABLE_ALL/DISABLE_ALL
; Parameters:
;   NONE
; Purpose:
;   These routines enable/disable all of the alarms.  It hides the disabled status of
;   the alarm by storing it in bit 3 of the alarm flags.
;      Bit0 = Alarm is enabled (SET=ENABLED)
;      Bit1 = Alarm is masked (SET=MASKED)
;      Bit2 = Current alarm is in 12 hour mode and is in the afternoon (SET=AFTERNOON)
;      Bit3 = Alarm was enabled, but we are hiding it (SET=HIDDEN)
;   It is safe to call these routine multiple times.
;---------------------------------------------------------------
ENABLE_ALL
ldx     #4                      ; We have 5 alarms to go through
ENABLE_NEXT
        lda     ALARM_STATUS,X          ; Get the flags for this alarm
        lsra                            ; Shift right 3 to get our hidden bit into place
lsra
lsra
        and     #1                      ; Mask out everything except the hidden bit (now in the enabled
position
        ora     ALARM_STATUS,X          ; Or it back into the flags
        and     #7                      ; and clear out our hidden bit
        sta     ALARM_STATUS,X          ; then save it out again.
        decx                            ; Count down the number of alarms
        bpl     ENABLE_NEXT             ; And go back for the next one
rts
DISABLE_ALL
ldx     #4                      ; We have 5 alarms to go through
DISABLE_NEXT
        lda     ALARM_STATUS,X          ; Get the flags for this alarm
        and     #1                      ; And extract our enabled bit
        lsla                            ; Shift left 3 to save as our hidden bit
lsla
lsla
        ora     ALARM_STATUS,X          ; Or it back into the flags
        and     #$0e                    ; and clear out the enabled bit
        sta     ALARM_STATUS,X          ; then save it out again.
        decx                            ; Count down the number of alarms
        bpl     DISABLE_NEXT            ; And go back for the next one
rts
```

This code has a few notable sections.

1. Program specific constants - We don't have any

2. System entry point vectors - This is where we have a lot of fun. We are using three of the entry points which we have never used before. The WRIST_DOTIC entry is enabled by us setting bit 3 in the Wristapp_flags which causes us to get called once a day. While we could enable it to call us hourly, by the minute, or even faster, it really doesn't make sense to waste processing time. The WRIST_INCOMM entry point gives us a chance to undo our hiding of the alarms just in case the downloaded data wants to mess with it. Lastly, the WRIST_NEWDATA entry is called after the data has been loaded into the watch.

3. Program strings - Of course we changed the strings once again. Note that the one string says WEEH and not WEEK since K is not a valid letter in the TIMEX6 alphabet. Don't worry, we will fix it up at runtime.

4. State Table(s) - We are back to only one state table. In fact, you will see that this state table is even less fancy than the hello world example. We really don't have any input functions, so we pretty much ignore everything.

5. State Table Handler0 - For state0, we only really need to handle the initial enter or resume where we put up the banner.

6. Faking the letter K - All we need to do is turn off one segment and turn on another to turn the H into a K.

7. Main Initialization routine - Nothing really significant here. This is called once when the wristapp is first loaded. We need to make sure that we set the appropriate bits in WRISTAPP_FLAGS. The new bit that we set here is to enable the callback once a day.

8. Determining the Current Day - This really is pretty simple, we figure out the current time zone and grab the day of the week from the right spot.

9. ENABLE_ALL/DISABLE_ALL - These routines are pretty simple also, all they have to do is hide the state of the enabled bit in the third bit of the alarm status flags. These routines had to be constructed so that you can call them many times in a row and not lose the original sense of the enabled bit for each alarm. We are able to do that by making sure that we always OR together the bits before clearing out the other.

Using 3 States - HexDump example

Ok, so you have a computer on your wrist. What better way to show it off than by having a hex dump utility to traipse through memory. This is a major overhaul of a previous version of the HexDump application that I have posted. I have turned it into a real application instead of a simple test program. It also uses the .ZSM file format to allow you to use it with ASM6805. You can download it here

```
;Name: Hex Dump
;Version: HEXDUMP
;Description: Hex Dumper - by John A. Toebes, VIII
;This Hex dump routine is a simple thing to test out dumping hex bytes...
;
; Press the NEXT/PREV buttons to advance/backup by 6 bytes of memory at a time
; Press the SET button to change the location in memory where you are dumping.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 0 indicates the direction of the last button
;   The other bits are not used
CURRENT_DIGIT   EQU     $62
DIGIT0          EQU     $63
DIGIT1          EQU     $64
DIGIT2          EQU     $65
DIGIT3          EQU     $66
;
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
```

```
L0123:  jmp     HANDLE_STATE0
        db      STATETAB0-STATETAB0
L0127:  jmp     HANDLE_STATE1
        db      STATETAB1-STATETAB0
L012b:  jmp     HANDLE_STATE2
        db      STATETAB2-STATETAB0
;
; (3) Program strings
;
S6_BYTE:        timex6   “ BYTE “
S6_DUMPER:      timex6  “DUMPER”
S8_LOCATION     Timex   “aaaa    “
;
; (4) State Table
;
STATETAB0:
        db      0
        db      EVT_ENTER,TIM2_12TIC,0          ; Initial state
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_TIMER2,TIM_ONCE,0           ; This is the timer
        db      EVT_DNNEXT,TIM2_8TIC,1          ; Next button
        db      EVT_DNPREV,TIM2_8TIC,1          ; Prev button
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_SET,TIM_ONCE,2              ; Set button
        db      EVT_USER0,TIM_ONCE,$FF          ; Return to system
        db      EVT_END

STATETAB1:
        db      0
        db      EVT_UPANY,TIM_ONCE,0            ; Releasing the prev or next button
        db      EVT_TIMER2,TIM2_TIC,1           ; Repeat operation with a timer
        db      EVT_END                         ; End of table

STATETAB2:
        db      2
        db      EVT_RESUME,TIM_ONCE,2           ; Resume from a nested app
        db      EVT_DNANY4,TIM_ONCE,2           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,2           ; NEXT, PREV, SET, MODE button released
        db      EVT_USER2,TIM_ONCE,0            ; Return to state 0
        db      EVT_END                         ; End of table

;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_ENTER                      ; Is this the initial state?
        bne     SHOWDATA                        ; no, just clean up the screen
```

```
;
; (6) Put up the initial banner screen
;
        jsr     CLEARALL                        ; Clear the display
        lda     #S6_BYTE-START                  ; Put ‘ BYTE ‘ on the top line
        jsr     PUT6TOP
        lda     #S6_DUMPER-START                ; Put ‘DUMPER’ on the second line
        jsr     PUT6MID
        lda     #SYS8_MODE                      ; Put MODE on the bottom line
        jmp     PUTMSGBOT
; (7) FMTHEX is a routine similar to FMTX, but it handles hex values instead
;========================================================================
; Routine: FMTHEX
; Purpose:
;   Format a byte into the buffer
; Parameters:
;   A - Byte to be formatted
;   X - Offset into Message buffer to put the byte
;========================================================================
FMTHEX:
        sta     S8_LOCATION,X   ; Save the byte
        and     #$0f            ; Extract the bottom nibble
        sta     S8_LOCATION+1,X ; Save the hex value of the nibble
        lda     S8_LOCATION,X   ; Get the value once again
        lsra                    ; Shift right by 4 to get the high order nibble
lsra
lsra
lsra

sta     S8_LOCATION,X   ; And put it back into the buffer
rts
;
; (8) This is called when we press the prev/next button or when the timer fires during that event
;
HANDLE_STATE1:
        lda     BTNSTATE
        cmp     #EVT_TIMER2                     ; Is this a repeat/timer event?
        beq     REPEATBTN                       ; yes, do as they asked

        bclr    0,FLAGBYTE                      ; Assume that they hit the prev button
        cmp     #EVT_DNPREV                     ; Did they hit the prev button
        bne     REPEATBTN                       ; Yes, we guessed right
        bset    0,FLAGBYTE                      ; No, they hit next.  Mark the direction.
REPEATBTN:
brclr   0,FLAGBYTE,NEXTLOC              ; If they hit the next button, go do that operation
;
; They pressed the prev button, let’s go to the previous location
;
PREVLOC:
        lda     CURRENT_LOC+1
        sub     #6
        sta     CURRENT_LOC+1
        lda     CURRENT_LOC
        sbc     #0
        sta     CURRENT_LOC
        bra     SHOWDATA
NEXTLOC:
        lda     #6
        add     CURRENT_LOC+1
        sta     CURRENT_LOC+1
        lda     CURRENT_LOC
        adc     #0
        sta     CURRENT_LOC
;
; (9) This is the main screen update routine.
; It dumps the current memory bytes based on the current address.  Note that since it updates the entire
; display, it doesn't have to clear anything
;
SHOWDATA:
jsr     CLEARSYM
clrx
        bsr     GETBYTE
        jsr     PUTTOP12

        ldx     #1
        bsr     GETBYTE
        jsr     PUTTOP34

        ldx     #2
        bsr     GETBYTE
        jsr     PUTTOP56

        ldx     #3
        bsr     GETBYTE
        jsr     PUTMID12

        ldx     #4
        bsr     GETBYTE
        jsr     PUTMID34

        ldx     #5
        bsr     GETBYTE
        jsr     PUTMID56

        lda     CURRENT_LOC             ; Get the high order byte of the address
clrx
        bsr     FMTHEX          ; Put that at the start of the buffer
        lda     CURRENT_LOC+1           ; Get the low order byte of the address
        ldx     #2
        bsr     FMTHEX          ; Put that next in the buffer

        lda     #S8_LOCATION-START
        jmp     BANNER8
```

```
; (10) GETBYTE gets a byte from memory and formats it as a hex value
;========================================================================
; Routine: GETBYTE
; Purpose:
;   Read a byte from memory and put it into DATDIGIT1/DATDIGIT2 as hex values
; Parameters:
;   X - Offset from location to read byte
;   CURRENT_LOC - Base location to read from
;========================================================================
GETBYTE
CURRENT_LOC     EQU     *+1                     ; Self modifying code... Point to what we want to modify
        lda     $4000,X                         ; Get the current byte
        sta     DATDIGIT2                       ; And save it away
        lsra                                    ; Extract the high nibble
lsra
lsra
lsra

        sta     DATDIGIT1                       ; And save it
        lda     DATDIGIT2                       ; Get the byte again
        and     #$0f                            ; Extract the low nibble
        sta     DATDIGIT2                       ; And save it
rts
;
; (11) State Table 2 Handler
; This is called to process the state events.
; We see SET, RESUME, DNANY4, and UPANY4 events
;
HANDLE_STATE2:
        bset    1,APP_FLAGS                    ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_UPANY4
        beq     REFRESH2
        cmp     #EVT_DNANY4                     ; Is this our initial entry?
        bne     FORCEFRESH
        lda     BTN_PRESSED                     ; Let’s see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_MODE                       ; Perhaps the MODE button
        beq     DO_MODE                         ; If so, handle it
; It must be the set button, so take us out of this state
        bsr     SHOWDATA
        lda     #EVT_USER2
        jmp     POSTEVENT
;
; (12) This handles the update routine to change a digit...
;
DO_NEXT
        bset    0,SYSFLAGS                      ; Mark our update direction as up
        bra     DO_UPD
```

```
DO_PREV
bclr    0,SYSFLAGS                      ; Mark our update direction as down
DO_UPD
clra
        sta     UPDATE_MIN                      ; Our low end is 0
        lda     #$F
        sta     UPDATE_MAX                      ; and the high end is 15 (the hes digits 0-F)
        bsr     GET_DISP_PARM
        lda     #UPD_DIGIT
        jsr     START_UPDATEP                   ; And prepare the update routine
        bset    4,BTNFLAGS                      ; Mark that the update is now pending
rts
;
; (13) This is where we switch which digit we are changing...
;
DO_MODE
lda     CURRENT_DIGIT
inca
        and     #3
        sta     CURRENT_DIGIT
;
; (14) Refresh the screen and start blinking the current digit...
;
REFRESH2
        lda     DIGIT0                          ; Get the first digit
        lsla                                    ; *16
lsla
lsla
lsla
        add     DIGIT1                          ; Plus the second digit
        sta     CURRENT_LOC                     ; To make the high byte of the address
        lda     DIGIT2                          ; Get the third digit
        lsla                                    ; *16
lsla
lsla
lsla
        add     DIGIT3                          ; Plus the fourth digit
        sta     CURRENT_LOC+1                   ; To make the low byte of the address
FORCEFRESH
        bclr    7,BTNFLAGS                      ; Turn off any update routine that might be pending
        jsr     SHOWDATA                        ; Format the screen
        ldx     #4                              ; We need to copy over 4 bytes from the buffer
COPYIT
        decx                                    ; This will be one down.
        lda     S8_LOCATION,X                   ; Get the formatted byte
        sta     DIGIT0,X                        ; And store it for the update routine
        tstx                                    ; Did we copy enough bytes?
        bne     COPYIT                          ; No, go back for more
        bsr     GET_DISP_PARM                   ; Get the parm for the blink routine
        lda     #BLINK_DIGIT                    ; Request to blink a digit
        jsr     START_BLINKP                    ; And do it
        bset    2,BTNFLAGS                      ; Mark a blink routine as pending
rts
;
; (15) This gets the parameters for an UPDATE/BLINK routine
;
GET_DISP_PARM
        lda     CURRENT_DIGIT                   ; Figure out what digit we are dumping
        sta     UPDATE_POS                      ; Store it for the BLINK/UPDATE routine
        add     #DIGIT0                         ; Point to the byte to be updated
        tax                                     ; And put it into X as needed for the parameter
rts
;
; (16) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     CURRENT_DIGIT                   ; Start out on the first digit
rts
```

This code has a few notable sections.

1. Program specific constants - We only really need special storage for the 4 digits which the update/blink routines will handle.

2. System entry point vectors - We only have a main. However, we also have 3 state tables.

3. Program strings - Nothing special here. We have two strings for the banner and one string that we show the current location with.

4. State Tables - We have three state tables now. State table0 does very little other than handle getting into states 1 and 2. State table 1 is for when you are pressing the prev/next buttons while in the main state to allow you to advance/backup by 6 bytes at a time. State Table 2 handles all of the setting of the digits. Note that it would be possible to combine these two states, but it would make the code much more complicated than it needs to be.

5. State Table 0 Handler - This is actually one of the simplest. All it has to do is put up the startup banner and then show the current data once that times out.

6. Initial banner screen - Very simple code to display the name of the application.

7. FMTHEX is a routine similar to FMTX, but it handles hex values instead. It is up here in order to allow several of the other BSR instructions to be able to reach the main update routine. Sometimes moving a subroutine can save you quite a few bytes.

8. PREV/NEXT Handling This is called when we press the prev/next button or when the timer fires during that event.

9. Main Update This is the main screen update routine. Note that we don't have to refresh anything since the entire screen is written.

10. GETBYTE gets a byte from memory and formats it as a hex value

11. State Table 2 Handler - This is very similar to the state handling in the passwd sample.
12. Changing Digits This handles the update routine to change a digit...
13. Switching Digits This is where we switch which digit we are changing...
14. Blinking Digits Refresh the screen and start blinking the current digit...
15. GET_DISP_PARM This gets the parameters for an UPDATE/BLINK routine. We made this a subroutine in order to ensure that everything is kept in sync. It also saves a few bytes.
16. Main Initialization This is the main initialization routine which is called when we first get the app into memory. As usual, there is not a lot that we have to do.

Dumping the EEPROM - promdump example

The HexDump program is great for dumping out the regular memory, but if you search and search, you will never find any of your appointments, lists, phone numbers, or anniversaries in the memory. That is because they are stored in an EEPROM outside of the address space. With a few simple modifications to the HexDump program, you can use the system to dump out the contents of the EEPROM. You can download it here

```
;Name: Prom Dump
;Version: promdump
;Description: Prom Dumper - by John A. Toebes, VIII
;This Prom Dump routine shows you what is in the EEProm
;
; Press the NEXT/PREV buttons to advance/backup by 6 bytes of memory at a time
; Press the SET button to change the location in memory where you are dumping.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE “WRISTAPP.I”
;
; (1) Program specific constants
;
FLAGBYTE        EQU     $61
;   Bit 0 indicates the direction of the last button
;   The other bits are not used
CURRENT_DIGIT   EQU     $62
DIGIT0          EQU     $63
DIGIT1          EQU     $64
DIGIT2          EQU     $65
DIGIT3          EQU     $66
;
; These should have been in the Wristapp.i files, but I forgot them...
;
INST_ADDRHI     EQU     $0437
INST_ADDRLO     EQU     $0438
HW_FLAGS        EQU     $9e
;
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  rts             ; Called to handle any timers or time events - WRIST_DOTIC
nop
nop
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
```

```
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB0-STATETAB0
L0127:  jmp     HANDLE_STATE1
        db      STATETAB1-STATETAB0
L012b:  jmp     HANDLE_STATE2
        db      STATETAB2-STATETAB0
;
; (3) Program strings
;
S6_EEPROM:      timex6   “EEPROM”
S6_DUMPER:      timex6  “DUMPER”
S8_LOCATION     Timex   “aaaa    “
;
; (4) State Table
;
STATETAB0:
        db      0
        db      EVT_ENTER,TIM2_12TIC,0          ; Initial state
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_TIMER2,TIM_ONCE,0           ; This is the timer
        db      EVT_DNNEXT,TIM2_8TIC,1          ; Next button
        db      EVT_DNPREV,TIM2_8TIC,1          ; Prev button
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_SET,TIM_ONCE,2              ; Set button
        db      EVT_USER0,TIM_ONCE,$FF          ; Return to system
        db      EVT_END

STATETAB1:
        db      0
        db      EVT_UPANY,TIM_ONCE,0            ; Releasing the prev or next button
        db      EVT_TIMER2,TIM2_TIC,1           ; Repeat operation with a timer
        db      EVT_END                         ; End of table

STATETAB2:
        db      2
        db      EVT_RESUME,TIM_ONCE,2           ; Resume from a nested app
        db      EVT_DNANY4,TIM_ONCE,2           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,2           ; NEXT, PREV, SET, MODE button released
        db      EVT_USER2,TIM_ONCE,0            ; Return to state 0
        db      EVT_END                         ; End of table

CURRENT_LOC
dw      $0000                           ; This is where we start in memory
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, TIMER2, and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_ENTER                      ; Is this the initial state?
        bne     SHOWDATA                        ; no, just clean up the screen
;
; (6) Put up the initial banner screen
;
        jsr     CLEARALL                        ; Clear the display
        lda     #S6_EEPROM-START                ; Put 'EEPROM' on the top line
        jsr     PUT6TOP
        lda     #S6_DUMPER-START                ; Put 'DUMPER' on the second line
        jsr     PUT6MID
        lda     #SYS8_MODE                      ; Put MODE on the bottom line
        jmp     PUTMSGBOT
; (7) FMTHEX is a routine similar to FMTX, but it handles hex values instead
;========================================================================
; Routine: FMTHEX
; Purpose:
;   Format a byte into the buffer
; Parameters:
;   A - Byte to be formatted
;   X - Offset into Message buffer to put the byte
;========================================================================
FMTHEX:
        sta     S8_LOCATION,X   ; Save the byte
        and     #$0f            ; Extract the bottom nibble
        sta     S8_LOCATION+1,X ; Save the hex value of the nibble
        lda     S8_LOCATION,X   ; Get the value once again
        lsra                    ; Shift right by 4 to get the high order nibble
lsra
lsra
lsra

sta     S8_LOCATION,X   ; And put it back into the buffer
rts
;
; (8) This is called when we press the prev/next button or when the timer fires during that event
;
HANDLE_STATE1:
        lda     BTNSTATE
        cmp     #EVT_TIMER2                     ; Is this a repeat/timer event?
        beq     REPEATBTN                       ; yes, do as they asked

        bclr    0,FLAGBYTE                      ; Assume that they hit the prev button
        cmp     #EVT_DNPREV                     ; Did they hit the prev button
        bne     REPEATBTN                       ; Yes, we guessed right
        bset    0,FLAGBYTE                      ; No, they hit next.  Mark the direction.
REPEATBTN:
```

```
brclr   0,FLAGBYTE,NEXTLOC              ; If they hit the next button, go do that operation
;
; They pressed the prev button, let's go to the previous location
;
PREVLOC:
        lda     CURRENT_LOC+1
        sub     #6
        sta     CURRENT_LOC+1
        lda     CURRENT_LOC
        sbc     #0
        sta     CURRENT_LOC
        bra     SHOWDATA
NEXTLOC:
        lda     #6
        add     CURRENT_LOC+1
        sta     CURRENT_LOC+1
        lda     CURRENT_LOC
        adc     #0
        sta     CURRENT_LOC
;
; (9) This is the main screen update routine.
; It dumps the current memory bytes based on the current address.  Note that since it updates the entire
; display, it doesn't have to clear anything
;
SHOWDATA:
jsr     CLEARSYM
clrx
        bsr     GETBYTE
        jsr     PUTTOP12

        ldx     #1
        bsr     GETBYTE
        jsr     PUTTOP34

        ldx     #2
        bsr     GETBYTE
        jsr     PUTTOP56

        ldx     #3
        bsr     GETBYTE
        jsr     PUTMID12

        ldx     #4
        bsr     GETBYTE
        jsr     PUTMID34

        ldx     #5
        bsr     GETBYTE
        jsr     PUTMID56

        lda     CURRENT_LOC             ; Get the high order byte of the address
```

```
clrx
        bsr     FMTHEX          ; Put that at the start of the buffer
        lda     CURRENT_LOC+1           ; Get the low order byte of the address
        ldx     #2
        bsr     FMTHEX          ; Put that next in the buffer

        lda     #S8_LOCATION-START
        jmp     BANNER8
; (10) GETBYTE gets a byte from memory and formats it as a hex value
;========================================================================
; Routine: GETBYTE
; Purpose:
;   Read a byte from memory and put it into DATDIGIT1/DATDIGIT2 as hex values
; Parameters:
;   X - Offset from location to read byte
;   CURRENT_LOC - Base location to read from
;========================================================================
GETBYTE
txa
        add     CURRENT_LOC+1
        sta     INST_ADDRLO
        lda     CURRENT_LOC
        adc     #0
        sta     INST_ADDRHI
        bset    6,HW_FLAGS                      ; Tell them that it is an EEPROM address
        jsr     GET_INST_BYTE                   ; Get the current byte
        sta     DATDIGIT2                       ; And save it away
        lsra                                    ; Extract the high nibble
lsra
lsra
lsra

        sta     DATDIGIT1                       ; And save it
        lda     DATDIGIT2                       ; Get the byte again
        and     #$0f                            ; Extract the low nibble
        sta     DATDIGIT2                       ; And save it
rts
;
; (11) State Table 2 Handler
; This is called to process the state events.
; We see SET, RESUME, DNANY4, and UPANY4 events
;
HANDLE_STATE2:
        bset    1,APP_FLAGS                    ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_UPANY4
        beq     REFRESH2
        cmp     #EVT_DNANY4                     ; Is this our initial entry?
        bne     FORCEFRESH
        lda     BTN_PRESSED                     ; Let’s see what the button they pressed was
        cmp     #EVT_PREV                       ; How about the PREV button
        beq     DO_PREV                         ; handle it
        cmp     #EVT_NEXT                       ; Maybe the NEXT button?
        beq     DO_NEXT                         ; Deal with it!
        cmp     #EVT_MODE                       ; Perhaps the MODE button
        beq     DO_MODE                         ; If so, handle it
; It must be the set button, so take us out of this state
        bsr     SHOWDATA
        lda     #EVT_USER2
        jmp     POSTEVENT
;
; (12) This handles the update routine to change a digit...
;
DO_NEXT
        bset    0,SYSFLAGS                      ; Mark our update direction as up
        bra     DO_UPD
DO_PREV
bclr    0,SYSFLAGS                      ; Mark our update direction as down
DO_UPD
clra
        sta     UPDATE_MIN                      ; Our low end is 0
        lda     #$F
        sta     UPDATE_MAX                      ; and the high end is 15 (the hes digits 0-F)
        bsr     GET_DISP_PARM
        lda     #UPD_DIGIT
        jsr     START_UPDATEP                   ; And prepare the update routine
        bset    4,BTNFLAGS                      ; Mark that the update is now pending
rts
;
; (13) This is where we switch which digit we are changing...
;
DO_MODE
lda     CURRENT_DIGIT
inca
        and     #3
        sta     CURRENT_DIGIT
;
; (14) Refresh the screen and start blinking the current digit...
;
REFRESH2
        lda     DIGIT0                          ; Get the first digit
        lsla                                    ; *16
lsla
lsla
lsla
        add     DIGIT1                          ; Plus the second digit
        sta     CURRENT_LOC                     ; To make the high byte of the address
        lda     DIGIT2                          ; Get the third digit
        lsla                                    ; *16
lsla
lsla
lsla
        add     DIGIT3                          ; Plus the fourth digit
        sta     CURRENT_LOC+1                   ; To make the low byte of the address
FORCEFRESH
        bclr    7,BTNFLAGS                      ; Turn off any update routine that might be pending
        jsr     SHOWDATA                        ; Format the screen
        ldx     #4                              ; We need to copy over 4 bytes from the buffer
COPYIT
        decx                                    ; This will be one down.
        lda     S8_LOCATION,X                   ; Get the formatted byte
        sta     DIGIT0,X                        ; And store it for the update routine
        tstx                                    ; Did we copy enough bytes?
        bne     COPYIT                          ; No, go back for more
        bsr     GET_DISP_PARM                   ; Get the parm for the blink routine
        lda     #BLINK_DIGIT                    ; Request to blink a digit
        jsr     START_BLINKP                    ; And do it
        bset    2,BTNFLAGS                      ; Mark a blink routine as pending
rts
;
; (15) This gets the parameters for an UPDATE/BLINK routine
;
GET_DISP_PARM
        lda     CURRENT_DIGIT                   ; Figure out what digit we are dumping
        sta     UPDATE_POS                      ; Store it for the BLINK/UPDATE routine
        add     #DIGIT0                         ; Point to the byte to be updated
        tax                                     ; And put it into X as needed for the parameter
rts
;
; (16) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                            ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
        clr     CURRENT_DIGIT                   ; Start out on the first digit
rts
```

This code is virtually identical to the promdump example with a few minor changes

1. Program specific constants - I didn't include these three important addresses in the Wristapp.i file, so you have to define them here.

2. System entry point vectors - No change.

3. Program strings - Of course we change the name of the application.

4. State Tables - No change here.

5. State Table 0 Handler - No change here.

6. Initial banner screen - No change here.

7. FMTHEX - No change here.

8. PREV/NEXT Handling - No change here.

9. Main Update - No change here.

10. GETBYTE This is the only real change. We have to call a system routine to read the byte from memory. Before we do that, we need to store the address into the INST_ADDR:HI_INST_ADDRLO variables and set the HW_FLAGS bit to indicate that it is an EEPROM address instead of a real memory address. Note that if we clear the bit instead of setting it, this program will behave like the HEXDUMP program.

11. State Table 2 Handler - No change here.

12. Changing Digits - No change here.

13. Switching Digits - No change here.

14. Blinking Digits - No change here.

15. GET_DISP_PARM - No change here.

16. Main Initialization - No change here.

Tracking Money - Spend Watch example

David Andrews [david@polarnet.com] gets the credit for the inspiration on this example. Of course it turned out to be a bit harder than I expected to write it - mostly due to the fact that I wanted it to be a full blown wristapp with lots of features yet still fit on the watch. This one also takes advantage of the 'parent' app which allows setting information in the applet without recompiling it.

What was the hardest about this application is making the user interface work and still be intuitive. Once I got past that, coding was just an exercise left to the reader.

There are a lot of tricks in this code to make it fit. I created a lot of subroutines and learned some interesting tricks to reduce code size. It currently sits at 713 bytes and I know how I can get 2 more bytes out of it, but I can't find much more fluff in the code to cut out. If you can find ways to make it smaller, I would be more than happy to hear about them...

You can download the wristapp and set program here

```
;Name: spend watch
;Version: spend0
;Description: spend watch - by John A. Toebes, VIII
;This keeps track of how much is in one of 7 categories
;
; Press the NEXT/PREV buttons to advance/backup through the categories
; Press the SET button to add/subtract/set/clear the amounts in the categories
; If you press the set button while the action is blinking, it will be carried out, otherwise
; you can cancel the operation.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
;Parent: SpendSet
;****************************************************************************************
;* Copyright © 1997 John A. Toebes, VIII                                              *
;* All Rights Reserved                                                                  *
;* This program may not be distributed in any form without the permission of the author *
;*         jtoebes@geocities.com                                                        *
;****************************************************************************************
;
; History:
;    31 July 96 - Corrected problem with totals not being recalculated when you reenter
;                 the wristapp.
;
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
; We use a few extra bytes here in low memory.  Since we can't possibly
; be running while the COMM app is running, we have no chance of
; conflicting with it's use of this memory.
;
BLINK_BUF       EQU     $5C     ; 3 Byte Buffer for the blink routine
```

```
;               EQU     $5D
;               EQU     $5E
CAT_SAVE        EQU     $5F     ; Temporary counter variable
COUNTER         EQU     $60     ; Temporary variable to hold the
FLAGBYTE        EQU     $61
;   Bit 0 indicates that the display does not need to be cleared
;   The other bits are not used

CURRENT_MODE    EQU     $62     ; The current mode that we are in
MODE_SELECT     EQU     0       ; Set mode, selecting which category to modify
MODE_HUNDREDS   EQU     1       ; Set mode, changing the hundreds of dollars digits
MODE_DOLLARS    EQU     2       ; Set mode, changing the dollars digits
MODE_CENTS      EQU     3       ; Set mode, changing the cents
MODE_ACTION     EQU     4       ; Set mode, changing the action
MODE_VIEW       EQU     5       ; Normal display mode

CATEGORY        EQU     $63     ; Current category
;
; These three bytes need to be contiguous.  The represent the current
; value that is being operated on
;
HUNDREDS        EQU     $64
DOLLARS         EQU     $65
CENTS           EQU     $66

ACTION          EQU     $67     ; Selector for the current action
ACT_ADD         EQU     0
ACT_SUB         EQU     1
ACT_SET         EQU     2
ACT_CLEAR       EQU     3
AMT_BASE        EQU     $F0
;
;
; (2) System entry point vectors
;
START   EQU     *
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  jmp     DO_UPD  ; Called to handle any timers or time events - WRIST_DOTIC
L0119:  rts             ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
nop
nop
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
nop
nop

L011f:  lda     STATETAB0,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE
```

```
        db      STATETAB0-STATETAB0
L0127:  jmp     HANDLE_STATE
        db      STATETAB1-STATETAB0
;
; (3) Program strings
;
; These strings represent the 4 possible actions.  They need to be early on in the data segment so that
; then can be pointed to by using 8-bit offset addressing.  They are exactly 3 bytes long and are
; displayed by using the BLINK_TZONE routine
;
S3_MODE:
S3_ADD          Timex   “ADD”
S3_SUB          Timex   “SUB”
S3_SET          Timex   “SET”
S3_CLR          Timex   “CLR”
;
; These are the categories that the end user has configured.  They are set by using the SPENDSET program
; which searches for the first string “TOTAL   “.  These strings must be exactly 8 bytes each in order
with
; total being the first one.
;
S8_TOTAL:       Timex   “TOTAL   “
S8_CAT1:        Timex   “CAT1    “
S8_CAT2:        Timex   “CAT2    “
S8_CAT3:        Timex   “CAT3    “
S8_CAT4:        Timex   “CAT4    “
S8_CAT5:        Timex   “CAT5    “
S8_CAT6:        Timex   “CAT6    “
S8_CAT7:        Timex   “CAT7    “
;
; These are the running amounts for each category.  Note that you can actually
; initialize them with some default and the code will run properly
;
AMT_TOTAL:      db      0,0,0
AMT_CAT1:       db      0,0,0
AMT_CAT2:       db      0,0,0
AMT_CAT3:       db      0,0,0
AMT_CAT4:       db      0,0,0
AMT_CAT5:       db      0,0,0
AMT_CAT6:       db      0,0,0
AMT_CAT7:       db      0,0,0
;
; These strings prompt for the current mode that we are in.  They are displayed on the top line of
; the display.
;
S6_SELECT       timex6  “SELECT”
S6_AMOUNT       timex6  “AMOUNT”
S6_ACTION       timex6  “ACTION”
S6_SPEND:       timex6  “SPEND”         ; save a byte by leaching off the space on the start of the next
string
S6_WATCH:       timex6  “ WATCH”
;
```

```
; This table selects which string is to be displayed.  It is directly indexed by the current mode
;
MSG_TAB         db      S6_SELECT-START ; 0 - MODE_SELECT
                db      S6_AMOUNT-START ; 1 - MODE_HUNDREDS
                db      S6_AMOUNT-START ; 2 - MODE_DOLLARS
                db      S6_AMOUNT-START ; 3 - MODE_CENTS
                db      S6_ACTION-START ; 4 - MODE_ACTION
                db      S6_SPEND-START  ; 5 - MODE_VIEW
;
; This is one of the magic tricks for providing the source for the blink routine.
; These are base pointers (offset from HUNDREDS) that we use to copy three bytes into
; BLINK_BUF.  The interesting one here is the MODE_CENTS entry which points to DATDIGIT1
; This works because the last number that we format happens to be the cents amount,
; and the blink routine expects the two characters instead of the actual value.
;
DATASRC         db      HUNDREDS-HUNDREDS       ; 1 - MODE_HUNDREDS
                db      DOLLARS-HUNDREDS        ; 2 - MODE_DOLLARS
                db      DATDIGIT1-HUNDREDS      ; 3 - MODE_CENTS
                db      S3_ADD-HUNDREDS         ; 4 - MODE_ACTION  0 - ACT_ADD
                db      S3_SUB-HUNDREDS         ; 4 - MODE_ACTION  1 - ACT_SUB
                db      S3_SET-HUNDREDS         ; 4 - MODE_ACTION  2 - ACT_SET
                db      S3_CLR-HUNDREDS         ; 4 - MODE_ACTION  3 - ACT_CLR
;
; This is the parameter to select which blink routine we want to use
;
BLINK_PARM      db      BLINK_MID12     ; 1 - MODE_HUNDREDS
                db      BLINK_MID34     ; 2 - MODE_DOLLARS
                db      BLINK_SECONDS   ; 3 - MODE_CENTS
                db      BLINK_TZONE     ; 4 - MODE_ACTION
;
; (4) State Tables
;
; This set of state tables is a little special since we actually use the
; same state processing routine for both states.  This saves us a lot of
; memory but still allows us to let the state table make it easy to exit
; the app with the MODE button
;
STATETAB0:
        db      0
        db      EVT_ENTER,TIM2_12TIC,0          ; Initial state
        db      EVT_RESUME,TIM_ONCE,0           ; Resume from a nested app
        db      EVT_TIMER2,TIM_ONCE,0           ; This is the timer
        db      EVT_MODE,TIM_ONCE,$FF           ; Mode button
        db      EVT_SET,TIM_ONCE,1              ; Set button
        db      EVT_DNANY4,TIM_ONCE,0           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_END

STATETAB1:
        db      1
        db      EVT_RESUME,TIM_ONCE,1           ; Resume from a nested app
        db      EVT_DNANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button pressed
        db      EVT_UPANY4,TIM_ONCE,1           ; NEXT, PREV, SET, MODE button released
        db      EVT_USER2,TIM_ONCE,0            ; Return to state 0
        db      EVT_END                         ; End of table
;
; (5) Put up the initial banner screen
;
HANDLE_ENTER
clra
        sta     CATEGORY                        ; We start out displaying the totals
        jsr     FETCH_CATEGORY
        jsr     CLEARALL                        ; Clear the display
        lda     #S6_SPEND-START                 ; Put ‘SPEND ‘ on the top line
        jsr     PUT6TOP
        lda     #S6_WATCH-START                 ; Put ‘ WATCH’ on the second line
        jsr     PUT6MID
        clr     FLAGBYTE                        ; Force us to clear the display
        lda     #MODE_VIEW                      ; Start out in the VIEW mode
        sta     CURRENT_MODE
        lda     #SYS8_MODE                      ; Put MODE on the bottom line
        jmp     PUTMSGBOT
;
; (6) This is the main screen update routine.
;---------------------------------------------------------------
; Routine:
;   SHOWCURRENT
; Parameters:
;   HUNDREDS,DOLLARS,CENTS - Current value to be displayed
;   0,FLAGBYTE - Screen state (CLR=Must clear it first)
;   CATEGORY - the current category to be displayed
; Returns:
;   DATDIGIT1,DATDIGIT2 - 2 digit characters for the cents value
; Purpose:
;   This routine shows the current selected category and value for the category
;---------------------------------------------------------------
SHOWCURRENT
        brset   0,FLAGBYTE,NOCLEAR              ; If we don’t need to clear the display, skip it
        jsr     CLEARALL                        ; Clear the display
        bset    0,FLAGBYTE                      ; And remember that we did it
NOCLEAR
        lda     #ROW_MP45                       ; Turn on the decimal point
        sta     DISP_ROW
        bset    COL_MP45,DISP_COL
        ldx     HUNDREDS                        ; Get the Hundreds
        jsr     FMTBLANK0                       ;   Format it
        jsr     PUTMID12                        ;   and display it
        ;
; We want to output the dollars, but if there were no hundreds, we want to let the
; first digit be a blank.  To do this, we simply let it be a blank and set it to a zero
; if there was actually anything in the hundreds field
        ;
        ldx     DOLLARS                         ; Get the Dollars
        jsr     FMTX                            ;   Format it
        tst     HUNDREDS                        ; Do we need to have a leading zero?
        beq     NOBLANKIT                       ; No, so it is fine
        ldx     DOLLARS                         ; Yes, Get the Dollars again
        jsr     FMTXLEAD0                       ;   And format it with a leading zero
NOBLANKIT
        jsr     PUTMID34                        ;   Display the Dollars
        ldx     CENTS                           ; Get the Cents
        jsr     FMTXLEAD0                       ;   Format it (and leave it around for later)
        jsr     PUTMID56                        ;   and display it.
        lda     CATEGORY                        ; Get which category we want
        lsla                                    ;  *2
        lsla                                    ;  *4
        lsla                                    ;  *8
        add     #S8_TOTAL-START                 ; *8+the start of the string
        jmp     BANNER8                         ;  and display the right string
;
; (7) State Table 0 and 1 Handler
; This is called to process the state events.
; We see SET, RESUME, DNANY4, and UPANY4 events
;
HANDLE_STATE:
        bset    1,APP_FLAGS                     ; Indicate that we can be suspended
        lda     BTNSTATE                        ; Get the event
        cmp     #EVT_ENTER                      ; Is this the initial state?
        beq     HANDLE_ENTER
        cmp     #EVT_DNANY4                     ; How about a button pressed?
        beq     HANDLE_DNANY
        bclr    1,BTNFLAGS                      ; Turn off the repeat counter
        cmp     #EVT_SET                        ; Did they press the set button
        bne     SKIP2                           ; No
        clr     CURRENT_MODE                    ; Yes, Go to MODE_SELECT
SKIP2   bra     GOREFRESH
;
; (8) They pressed a button, so handle it
;
HANDLE_DNANY
        lda     BTN_PRESSED                     ; Let's see what the button they pressed was
        beq     DO_NEXT                         ; MODE=1, and NEXT=0, so if it is less, it must be the
next button
        cmp     #EVT_SET                        ; MODE=1 SET=2 PREV=3, test all at once
        blo     DO_MODE                         ; <2 = 1 so we have a EVT_MODE
        bhi     DO_PREV                         ; >2 = 3 so we have a EVT_PREV
        ;
        ; They pressed the set button, so we want to carry out the operation IF they have
        ; one currently selected.
        ;
DO_SETOUT
        lda     CURRENT_MODE                    ; See what mode we were in
        cmp     #MODE_ACTION                    ; Is it the ACTION mode?
        bne     NO_ACTION                       ; No, so just cancel the operation
        jsr     DO_OPERATION                    ; Do what they requested
        jsr     DO_TOTAL                        ; And total up everything
        jsr     PLAYCONF                        ; Plus tell them that we did it
NO_ACTION
        bclr    0,FLAGBYTE                      ; We need to clear the display
        lda     #MODE_VIEW                      ; And switch back to VIEW mode
        sta     CURRENT_MODE
        lda     #EVT_USER2                      ; And go back to state 0
        jmp     POSTEVENT
;
; (9) This handles the update routine to change a digit...
;
DO_NEXT
        bset    0,SYSFLAGS                      ; Mark our update direction as up
        BRSKIP2                                 ; and skip over the next instruction
DO_PREV
        bclr    0,SYSFLAGS                      ; Mark our update direction as down
DO_UPD
        lda     CURRENT_MODE                    ; Which mode are we in?
        beq     CHANGE_CATEGORY                 ; 0=MODE_SELECT, so change the category
        cmp     #MODE_VIEW                      ; 5=MODE_VIEW, so we also change the category
        bne     TRYOTHERS
CHANGE_CATEGORY
; (10) updating the category
        ldx     #CATEGORY                       ; Point to the category variable
        lda     #7                              ; get our range of values
        bsr     ADJUST_PX_ANDA                  ; And let the routine do the adjust for us
        jsr     FETCH_CATEGORY                  ; Update the current amount from the new category
GOREFRESH
        bra     REFRESH
;
; (11) ADJUST_PX_ANDA - a routine to adjust a value based on the direction
;-------------------------------------------------------------
; Routine:
;   ADJUST_PX_ANDA
; Parameters:
;   A - Binary range to limit value within ((2**x)-1)
;   0,SYSFLAGS - Direction to adjust, SET=UP
;   X - Pointer to value to be adjusted
; Returns:
;   Value pointed to by X is adjusted
; Purpose:
;   This routine adjusts a value up or down based on the current direction, wrapping
;   it to the binary range indicated by the value in A.  Note that this value must
;   be a power of 2-1 (e.g. 1, 3, 7, 15, 31, 63, or 127)
;-------------------------------------------------------------
ADJUST_PX_ANDA
        inc     ,X
        brset   0,SYSFLAGS,NODEC
        dec     ,X
        dec     ,X
NODEC   and     ,X
        sta     ,X
        rts
;
; (12) Try updating one of the other modes
; We have already handled MODE_SELECT and MODE_VIEW.  This code handles
; MODE_HUNDREDS, MODE_DOLLARS, MODE_CENTS, and MODE_ACTION
;
TRYOTHERS
        cmp     #MODE_CENTS                     ; 3=MODE_CENTS
        bls     TRYMORE                         ; If it is <=, then we leave only MODE_ACTION
; (13) updating the Action
        lda     CATEGORY                        ; Which category is it?
        beq     REFRESH                         ; If we are displaying the total, you can't change the
action
        ldx     #ACTION                         ; Point to the current action
        lda     #3                              ; and the range of actions
        bsr     ADJUST_PX_ANDA                  ; and let our simple routine handle it for us
        bra     REFRESH
TRYMORE
        beq     DOCENTS                         ; If it is MODE_CENTS, go handle it
; (14) Update MODE_HUNDREDS=1 and MODE_DOLLARS=2
        clrx                                    ; Set the lower limit =0
        stx     UPDATE_MIN
        ldx     #99                             ; And the upper limit= 99
        stx     UPDATE_MAX
        add     #HUNDREDS-1                     ; Point to the right byte to update
        tax                                     ; And put it in X as the parameter
        lda     CURRENT_MODE                    ; MODE=1       MODE=2
        deca                                    ;    0            1
        lsla                                    ;    0            2
        add     #UPD_MID12                      ;  5=UPD_MID12 7=UPD_MID34
        jsr     START_UPDATEP                   ; And prepare the update routine
        bset    4,BTNFLAGS                      ; Mark that the update is now pending
        rts
;
; (15) This is where we switch which digit we are changing...
;
DO_MODE
        lda     CURRENT_MODE                    ; Get the mode
        ldx     #MODE_ACTION                    ; Limit it to the first 5 modes
        jsr     INCA_WRAPX                      ; And let the system increment it for us
        sta     CURRENT_MODE                    ; Save it back
        ; When we switch to the ACTION mode and we have the Totals category showing,
        ; we need to limit them to the single action of CLEAR
        ;
        cmp     #MODE_ACTION                    ; Did we go to action mode?
        bne     REFRESH                         ; No, nothing to do
        clr     ACTION                          ; Reset the action to be add
        tst     CATEGORY                        ; Are we displaying the totals
        bne     REFRESH                         ; No, nothing more to do
        lda     #ACT_CLEAR                      ; Yes, switch them to CLEAR
        sta     ACTION
```

```
;
; (16) Refresh the screen and start blinking the current digit...
;
REFRESH
        ; 0 - SELECT   <Category>
        ; 1 - AMOUNT    (Blink hundreds)
        ; 2 - AMOUNT    (Blink dollars)
        ; 3 - AMOUNT    (Blink cents)
        ; 4 - ACTION
        jsr     SHOWCURRENT                     ; Format the screen
        ldx     CURRENT_MODE                    ; Get the mode
        lda     MSG_TAB,X                       ; So that we can get the message for it
        jsr     PUT6TOP                         ; And put that on the top of the display
        ;
        ; Now we need to make the right thing blink
        ;
        ldx     CURRENT_MODE                    ; Are we in Select mode?
        beq     NOBLINK2                        ; Yes, don't blink anything
        cpx     #MODE_ACTION                    ; How about ACTION MODE?
        bhi     NOBLINK2                        ; >ACTION is VIEW mode, so if so, don't blink either
        ; 1 -> BLINK_MID12    PARM=&HUNDREDS
        ; 2 -> BLINK_MID34    PARM=&DOLLARS
        ; 3 -> BLINK_SECONDS  PARM=&2Characters
        ; 4 -> BLINK_TZONE    PARM=&3Characters
        brset   1,BTNFLAGS,NOBLINK2             ; Also, we don't want to be blinking if we are in an
update routine
        bne     SETUP_BLINK                     ; If we were not in action mode, we have the right data
source
        ; Put a > on the display
        ldx     #C_RIGHTARR                     ; Put a > sign right in front of the action
        lda     #POSL3_5
        jsr     PUTLINE3
        lda     CURRENT_MODE                    ; Get the mode
        add     ACTION                          ; And add in the action
        tax                                     ; To compute our data source pointer
SETUP_BLINK
;
; (17) Set up the parameters for and call the blink routine
;
        ldx     DATASRC-1,X                     ; Get the offsetted pointer to the right data
        lda     HUNDREDS,X                      ; And copy the 3 bytes to our blink buffer
        sta     BLINK_BUF
        lda     HUNDREDS+1,X
        sta     BLINK_BUF+1
        lda     HUNDREDS+2,X
        sta     BLINK_BUF+2
        ldx     CURRENT_MODE                    ; Get our mode again
        lda     BLINK_PARM-1,X                  ; and use it to pick up which parameter we are passing
        ldx     #BLINK_BUF                      ; Point to the common blink buffer
        jsr     START_BLINKP                    ; And do it
        bset    2,BTNFLAGS                      ; Mark a blink routine as pending
NOBLINK2
```

```
rts
;
; (18) Update MODE_CENTS
; This is a special case since we don't have a system routine that allows updating
; the right most digits on the middle line.  Fortunately we can fake it by turning
; on the tic timer and waiting until 8 tics have passed before going into a repeat
; loop.  The code has been carefully constructed so that the tic timer can just go
; straight to the DO_UPD code to work.
DOCENTS
        ldx     #COUNTER                        ; Point to the counter (saves code size)
        brset   1,BTNFLAGS,NOSTART              ; Are we already in an update loop?
        lda     #8                              ; No, we need to wait 8 tics
        sta     ,X      ; X->COUNTER            ; Save the value
        BSET    1,BTNFLAGS                      ; and start the timer
        bra     DOIT                            ; But still do it once right now
;
DEC_DELAY
        dec     ,X      ; X->COUNTER            ; We haven't hit the limit, decrement it and try again
        rts
NOSTART
        tst     ,X      ; X->COUNTER            ; We are in the loop, have we hit the limit?
        bne     DEC_DELAY                       ; no, go off and delay once more
DOIT
        lda     #99                             ; Our upper limit is 99
        ldx     #CENTS                          ; Point to the cents variable (saves code size)
        brset   0,SYSFLAGS,UPCENTS              ; Are we in an up mode?
        dec     ,X      ; X->CENTS              ; Down, decrement the value
        bpl     REFRESH                         ; If we didn't wrap, just go display it
        sta     ,X      ; X->CENTS              ; We wrapped, save the upper limit
        bra     REFRESH                         ; and go display it
UPCENTS
        inc     ,X      ; X->CENTS              ; Up, increment the value
        cmp     ,X      ; X->CENTS              ; Did we hit the limit?
        bpl     REFRESH                         ; No, go display it
        clr     ,X      ; X->CENTS              ; Yes, wrap to the bottom
        bra     REFRESH                         ; and display it
;
; (19) DO_OPERATION - Perform the requested operation
;----------------------------------------------------------------
; Routine:
;   DO_OPERATION
; Parameters:
;    HUNDREDS,DOLLARS,CENTS - Amount to be added/subtracted/set
;    CATEGORY - Item to be updated
;    ACTION - 0 = ACT_ADD
;             1 = ACT_SUB
;             2 = ACT_SET
;             3 = ACT_CLEAR
; Purpose:
;   Adjusts the corresponding category by the given amount
;----------------------------------------------------------------
```

```
DO_OPERATION
        lda     CATEGORY                        ; Get our category
        bsr     COMPUTE_BASE                    ; And point to the data for it
        lda     ACTION                          ; Which action is it?
        beq     DO_ADD                          ; 0=ADD, go do it
        cmp     #ACT_SET                        ; 3 way compare here... (code trick)
        beq     DO_SET                          ; 2=SET, go do it
        blo     DO_SUB                          ; <2=1 (SUB), go do it
DO_CLR                                          ; >2 = 3 (CLEAR)
        clr     HUNDREDS                        ; Clear out the current values
        clr     DOLLARS
        clr     CENTS
        tst     CATEGORY                        ; Were we clearing the total?
        bne     DO_SET                          ; No, just handle it
        ;
; They want to clear everything
        ;
        ldx     #(3*8)-1                        ; Total number of categories
CLEAR_TOTALS
; Mini Routine here X=number of bytes to clear
        clra
CLR_MORE
        sta     AMT_TOTAL,X                     ; Clear out the next byte
        decx                                    ; Decrement the number to do
        bpl     CLR_MORE                        ; And go for more
        rts
;
; (20) Handle Subtracting a value
;
DO_SUB
        neg     HUNDREDS                        ; Just negate the value to be added
        neg     DOLLARS
        neg     CENTS                           ; And fall into the add code
;
; (21) Handle Adding a value
;
DO_ADD
        lda     CENTS                           ; Add the cents
        add     AMT_BASE+2,X
        sta     CENTS
        lda     DOLLARS                         ; Add the dollars
        add     AMT_BASE+1,X
        sta     DOLLARS
        lda     HUNDREDS                        ; Add the hundreds
        add     AMT_BASE,X
        sta     HUNDREDS
        ldx     #CENTS                          ; Point to the cents as it will be the first one we fix up
        tst     ACTION                          ; See what type of operation we just did
        beq     FIXUP_ADD                       ; Was it an ADD? If so, do do it
        bsr     TRYDEC                          ; Decrement, fix up the Cents
        bsr     TRYDEC                          ; Then fix up the dollars
        lda     HUNDREDS                        ; Did the hundreds underflow as a result?
        bmi     DO_CLR                          ; Yes, so just set everything to zero
        bra     DO_SET                          ; No, so copy over the values to the current entry
TRYDEC
        lda     ,X                              ; Get the current byte to check
        bpl     RETDEC                          ; If it didn't underflow, then skip to the next byte
        add     #100                            ; Add back the 100 that it underflowed
        sta     ,X                              ; And save that away
        decx                                    ; Back up to the next most significant byte
        dec     ,X                              ; and borrow the one
        rts
RETDEC  decx                                    ; No need to do anything, so skip to the next byte
        rts
TRYADD
        lda     ,X                              ; Get the current byte to check
        sub     #100                            ; See if it was less than 100
        bmi     RETDEC                          ; If so, then it was already normalized so skip out
        sta     ,X                              ; It was an overflow, so save the fixed value
        decx                                    ; Skip to the next byte
        inc     ,X                              ; And add in the overflow
        rts
FIXUP_ADD
        bsr     TRYADD                          ; Fix up the cents
        bsr     TRYADD                          ; and then fix up the dollars
;
; (22) Handle setting a value
;
DO_SET
        bsr     COMPUTE_CATEGORY_BASE           ; Point to the data for our category
        lda     HUNDREDS                        ; Copy over the values to the current category
        sta     AMT_BASE,X
        lda     DOLLARS
        sta     AMT_BASE+1,X
        lda     CENTS
        sta     AMT_BASE+2,X
        rts
;
; (23) COMPUTE_BASE - Computes an offset pointer to get to the total amounts
; This is a trick to save us a few bytes in the instructions.
;---------------------------------------------------------------
; Routine:
;   COMPUTE_BASE
; Parameters:
;   A - Offset into total
; Returns:
;   X - Pointer relative to AMT_BASE to use
; Purpose:
;   Computes an offset pointer to get to the total amounts
;---------------------------------------------------------------
COMPUTE_CATEGORY_BASE
      lda     CATEGORY                        ; Get our category
COMPUTE_BASE
```

```
        ldx     #3
        mul
        add     #AMT_TOTAL-AMT_BASE
        tax
        rts
;
; (24) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0                        ; We want button beeps and to indicate that we have been
loaded
        sta     WRISTAPP_FLAGS
; Fall into DO_TOTAL
;
; (25) DO_TOTAL - Recomputes the current total
;----------------------------------------------------------------
; Routine:
;   DO_TOTAL
; Parameters:
;   NONE
; Purpose:
;   Recomputes the current total
;----------------------------------------------------------------
DO_TOTAL
        lda     CATEGORY                    ; Remember our category
        sta     CAT_SAVE
        clr     ACTION                      ; Say that we want to add 0=ACT_ADD
        clr     CATEGORY                    ; To the total category
        ldx     #2                          ; But we need to clear it first
        bsr     CLEAR_TOTALS
        lda     #7                          ; And iterate over the 7 categories
        sta     COUNTER
TOT_LOOP
        lda     COUNTER                     ; Get our current category
        bsr     FETCH_CATEGORY              ; And fetch the data
        jsr     DO_OPERATION                ; Then add it to the total
        dec     COUNTER                     ; Go to the next category
        bne     TOT_LOOP                    ; Until we are done
        lda     CAT_SAVE                    ; Restore the category
        sta     CATEGORY
; fall into FETCH_CATEGORY
; (26) FETCH_CATEGORY - Retrieves the value of the total amount for the selected category
;----------------------------------------------------------------
; Routine:
;   FETCH_CATEGORY
; Parameters:
;   A - Category to be fetched
; Returns:
;   HUNDREDS,DOLLARS,CENTS - Current value of selected category
; Purpose:
;   Retrieves the value of the total amount for the selected category
;----------------------------------------------------------------
```

```
FETCH_CATEGORY
        bsr     COMPUTE_BASE                    ; Get the pointer to the base
        lda     AMT_BASE,X                      ; And retrieve the data
        sta     HUNDREDS
        lda     AMT_BASE+1,X
        sta     DOLLARS
        lda     AMT_BASE+2,X
        sta     CENTS
        rts
;--------------------END OF CODE---------------------------------------------------
```

This is a pretty significant program and the sections are ordered to make the branches all work out. Here's a quick look around at the sections.

1. Program specific constants - It is worth noting that in this case, I actually intruded on the space which one might consider reserved for the system applications. However, the only one that uses any of this memory is the Comm app and there is no chance that we need to be running while it is. We are forced in several instances to use this lower memory because the system roms need a pointer passed in X. Since our code loads into 0110 and beyond, we have to use lower memory if we want to actually point to something.

2. System entry point vectors - Nothing really special here. However, we do have a timer routine that we enable when we are inputting cents. What is nice in this case is that the code is constructed so that it jumps right into the processing loop to act as if a timer event had occurred with the normal state processing.

3. Program strings - We have quite a few strings that we have created. We also take advantage of table of pointers to save us code space.

4. State Tables - This is a pretty unusual program in that even though we have two state tables, they both point to the same state table processing routine. This allows me to let the system handle knowing when we are in set mode to allow for the mode button to advance us through states in the set mode and to take us out of the wristapp when we are not in set mode.

5. Initial Banner Screen - No real surprises here.

6. This is the main screen update routine.

7. State Table 0 and 1 Handler

8. They pressed a button, so handle it

9. This handles the update routine to change a digit...

10. updating the category

11. ADJUST_PX_ANDA - a routine to adjust a value based on the direction

12. Try updating one of the other modes

13. updating the Action

14. Update MODE_HUNDREDS=1 and MODE_DOLLARS=2

Creating a Sound Scheme - Sound1 example

With a little prodding, I decided to update the assembler so that allows you to create a sound scheme automatically. This is a very simple sound scheme which gives you the same sounds as the Datalink default ones. Use this as a basis to create any new ones that you might want.

```
;Sound: Datalink Default
;Version: Sound1
;
; This sample corresponds to the default sounds that you get when you reset a DataLink
; watch to its default state.
;
;****************************************************************************************
;* Copyright © 1997 John A. Toebes, VIII                                              *
;* All Rights Reserved                                                                  *
;* This program may not be distributed in any form without the permission of the author *
;*         jtoebes@geocities.com                                                        *
;****************************************************************************************
;
            INCLUDE “WRISTAPP.I”
;
; This is the default sound table
;
DEF_SOUNDS
        db      SP_1-SD_1       ; 0000: 08

        db      SD_1-DEF_SOUNDS ; 0001: 0b   BUTTON BEEP
        db      SD_2-DEF_SOUNDS ; 0002: 0c   RETURN TO TIME
        db      SD_3-DEF_SOUNDS ; 0003: 0d   HOURLY CHIME
        db      SD_4-DEF_SOUNDS ; 0004: 0e   CONFIRMATION
        db      SD_5-DEF_SOUNDS ; 0005: 0f   APPOINTMENT BEEP
        db      SD_5-DEF_SOUNDS ; 0006: 0f   ALARM BEEP
        db      SD_5-DEF_SOUNDS ; 0007: 0f   PROGRAM DOWNLOAD
        db      SD_5-DEF_SOUNDS ; 0008: 0f   EXTRA
        db      SD_6-DEF_SOUNDS ; 0009: 11   COMM ERROR
        db      SD_7-DEF_SOUNDS ; 000a: 12   COMM DONE
;
; This is the soundlet count table which contains the duration
; counts for the individual soundlets
;
SD_1    db      SND_END+1       ; 000b: 81
SD_2    db      SND_END+1       ; 000c: 81
SD_3    db      SND_END+2       ; 000d: 82
SD_4    db      SND_END+4       ; 000e: 84
SD_5    db      10,SND_END+40   ; 000f: 0a a8
SD_6    db      SND_END+10      ; 0011: 8a
SD_7    db      SND_END+32      ; 0012: a0
;
; This is the soundlet pointer table which contains the pointers to the soundlets
;
SP_1    db      SL_2-DEF_SOUNDS ; 0013: 1d
```

```
SP_2    db      SL_1-DEF_SOUNDS ; 0014: 1b
SP_3    db      SL_3-DEF_SOUNDS ; 0015: 1f
SP_4    db      SL_2-DEF_SOUNDS ; 0016: 1d
SP_5    db      SL_4-DEF_SOUNDS ; 0017: 22
        db      SL_5-DEF_SOUNDS ; 0018: 27
SP_6    db      SL_6-DEF_SOUNDS ; 0019: 2a
SP_7    db      SL_2-DEF_SOUNDS ; 001a: 1d
;
; These are the soundlets themselves.  The +1 or other number
; indicates the duration for the sound.
;
SL_1    db      TONE_HI_GSHARP+1                ; 001b: 91
        db      TONE_END                        ; 001c: 00

SL_2    db      TONE_MID_C+1                    ; 001d: 31
        db      TONE_END                        ; 001e: 00

SL_3    db      TONE_MID_C+2                    ; 001f: 32
        db      TONE_PAUSE+2                    ; 0020: f2
        db      TONE_END                        ; 0021: 00

SL_4    db      TONE_HI_C+2                     ; 0022: 22
        db      TONE_PAUSE+2                    ; 0023: f2
        db      TONE_HI_C+2                     ; 0024: 22
        db      TONE_PAUSE+10                   ; 0025: fa
        db      TONE_END                        ; 0026: 00

SL_5    db      TONE_HI_C+2                     ; 0027: 22
        db      TONE_PAUSE+2                    ; 0028: f2
        db      TONE_END                        ; 0029: 00

SL_6    db      TONE_HI_C+3                     ; 002a: 23
        db      TONE_MID_C+3                    ; 002b: 33
        db      TONE_END                        ; 002c: 00
;
; This is the tone that the comm app plays for each record
;
        db      TONE_MID_C/16                    ; 002d: 03
```

Random Numbers and Marquis - 3Ball example

Wayne Buttles contributed the first version of this Wristapp which gives you a simple decision maker. It inspired me to make a few adjustments to it and add a real random number generator that you can use. I've also included a little busy wait Marquis while it is selecting a number to show off a use of the time. This Wristap also illustrates that you don't always have to put a JMP or RTS instruction in the entry point vectors.

```
;Name: 3BALL
;Version: 3BALL
;Description: An executive decision maker that will give a yes/no/maybe answer.  Pressing Next will
generate another answer and beep (since it will be the same answer sometimes).
;
;© 1997 Wayne Buttles (timex@fdisk.com). Compiled using tools and knowledge published by John A. Toebes,
VIII and Michael Polymenakos (mpoly@panix.com).
; Some enhancements by John Toebes...
;
;HelpFile: watchapp.hlp
;HelpTopic: 100
;
; (1) Program specific constants
;
INCLUDE "WRISTAPP.I"
;
; Program specific constants
;
CURRENT_TIC     EQU     $27     ; Current system clock tic (Timer)
LAST_ANS        EQU     $61
RAND_SEED       EQU     $60
MARQ_POS        EQU     $62
START           EQU    *
;
; (2) System entry point vectors
;
L0110:  jmp     MAIN          ; The main entry point - WRIST_MAIN
L0113:  bclr    1,BTNFLAGS    ; Called when we are suspended for any reason - WRIST_SUSPEND
        rts
L0116:  jmp     FLASH         ; Called to handle any timers or time events - WRIST_DOTIC
L0119:  bclr    1,BTNFLAGS    ; Called when the COMM app starts and we have timers pending - WRIST_INCOMM
        rts
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
        nop
        nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
        rts
L0123:  jmp   HANDLE_STATE0
        db    STATETAB-STATETAB
;
; (3) Program strings
;
S6_MSG  timex6  "3 BALL"
```

```
S6_MAYBE timex6 “MAYBE”
S6_YES  timex6  “ YES”
S6_NO   timex6  “  NO”
S6_MARQ timex6  “   +O+   “

MARQ_SEL
        DB      S6_MARQ+2-START
        DB      S6_MARQ+3-START
        DB      S6_MARQ+2-START
        DB      S6_MARQ+1-START
        DB      S6_MARQ-START
        DB      S6_MARQ+1-START

MSG_SEL DB      S6_YES-START
        DB      S6_NO-START
        DB      S6_MAYBE-START
        DB      S6_YES-START
;
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM2_16TIC,0  ; Initial state
        db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
        db      EVT_DNNEXT,TIM2_16TIC,0 ; Next button
        db      EVT_TIMER2,TIM_ONCE,0   ; Timer
        db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, RESUME, TIMER2 and NEXT events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS             ; Indicate that we can be suspended
        bclr    1,BTNFLAGS              ; Turn off the MARQUIS tic event
        lda     BTNSTATE
        cmp     #EVT_DNNEXT             ; Did they press the next button?
        beq     DOITAGAIN
        cmp     #EVT_ENTER              ; Or did we start out
        beq     DOITAGAIN
        cmp     #EVT_RESUME
        beq     REFRESH
;
; (6) Select a random answer
;
SHOWIT
        bsr     RAND
        and     #3              ; go to a 1 in 4 chance
        sta     LAST_ANS
;
; (7) Display the currently selected random number
```

```
;
REFRESH
        ldx     LAST_ANS        ; Get the last answer we had, and use it as an index
        lda     MSG_SEL,X       ; And get the message to display
        jsr     PUT6TOP         ; Put that on the top
BANNER
        lda     #S6_MSG-START
        jsr     PUT6MID
        lda     #SYS8_MODE      ; And show the mode on the bottom
        jmp     PUTMSGBOT
;
; (8) This flashes the text on the screen
;
FLASH
        lda     CURRENT_APP     ; See which app is currently running
        cmp     #APP_WRIST      ; Is it us?
        bne     L0113           ; No, so just turn off the tic timer since we don't need it
        ldx     #5
        lda     MARQ_POS
        jsr     INCA_WRAPX
        sta     MARQ_POS
        tax
        lda     MARQ_SEL,X
        jmp     PUT6TOP
;
; (9) They want us to do it again
;
DOITAGAIN                       ; Tell them we are going to do it again
        clr     MARQ_POS
        bset    1,BTNFLAGS
        bra     BANNER
;
; (10) Here is a simple random number generator
;
RAND
        lda     RAND_SEED
        ldx     #85
        mul
        add     #25
        sta     RAND_SEED
        rola
        rola
        rola
        rts
;
; (11) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0            ; We want button beeps and to indicate that we have been loaded
        sta     WRISTAPP_FLAGS
        lda     CURRENT_TIC
        sta     RAND_SEED
        rts
```

1. Program specific constants - We have two variables - RAND_SEED and CURRENT_TIC which we use for the random number routine. RAND_SEED is used to keep track of the last random number returned so that we continue to deliver random numbers. CURRENT_TIC is what is set by the system when it reads the clock to keep the watch time up to date. We use it once to provide a seed for the random number generator.

2. System entry point vectors - This one gets to be a little fun. Notice for the WRIST_SUSPEND and WRIST_INCOMM routines that we don't have a JMP instruction, but instead put the actual code in line. This saves use a couple of bytes.

3. Program strings - We are pretty frugal here in reusing blanks at the end of the string very liberally. Also note the S6_MARQ string which has blanks at the start and end so that it can shuffle left and right on the display but always have blanks visible. The MARQ_SEL and MSG_SEL tables are simply offsets that allow us to select the message with a simple load instruction instead of having to calculate the offset.

4. State Table - This is pretty vanilla here except for the fact that we have a very long time interval after the DNNEXT and ENTER events. It is during this time that the Marquis runs. We could make it even longer, but this seems to be a good compromise between seeing something happen and actually getting a result in a reasonable time.

5. State Table 0 Handler - Extremely simple, there are only four events that we want to see and this is the typical test and branch one. The only unique thing here is that we turn off the Marquis timer as soon as we get any event.

6. Select a random answer - As if life weren't complicated enough. This is where we go when it is time to make a decision. For this we get a random number and limit it to 1 in four.

7. Display the currently selected random number - Given a random number, we just get the message for it and put it on the display.

8. This flashes the text on the screen - This is the cheap way to do a Marquis. Just have a string wider than the display and change the offset from the start at which you start to display. For this one, there are only 6 states and we select the starting offset from the table based on our current cycle. Note that this routine is called by the TIC timer which is enabled when they want a new random number. Eventually the timer for the main event will run out and they will simply stop calling us.

9. They want us to do it again - Whenever we want to do a new random number, we just start the Marquis tic timer and set up the display.

10. Here is a simple random number generator - This is a random number generator that you might want to use. It is a derivative of the typical calculation rand = (seed*25173 + 13849) MOD 65536 which I have chopped down to fit in the 8 bit world as rand = (seed * 85 + 25) MOD 256. Because the low order bits do produce a pattern cycle which is fairly predictable, we rotate through to get a few of the more randomly occurring bits.

11. This is the main initialization routine which is called when we first get the app into memory - Very boring stuff here, but we do take a moment to initialize the random number seed with the current tic count just to make it a little more variable.

Playing Hourly Chimes - Ships Bells example

Theron E. White, CPA" <twhite@mercury.peganet.com> suggested a wristapp to allow the hourly chimes to play the number of bells past a shift change. This would be 8 bells at midnight, 8AM, and 4PM, 1 bell at 1AM, 9AM, and 5PM, with one more bell for each hour after that. This wristapp is a little unique in that it doesn't use the sound playing routines directly, but instead goes straight to the hardware. This allows you to have whatever sound scheme you want in the watch. The pattern for the bells and the actual tone is customizable below. This app is also a good candidate for combining with another wristapp as this one has no real user input operations.

```
;Name: Ships Bells
;Version: SHIPBELL
;Description: Ships bells - by John A. Toebes, VIII
;This application turns makes the hour chime with nautical bells.
;
;TIP:  Download your watch faster:  Download a WristApp once, then do not send it again.  It stays in the
watch!
;HelpFile: watchapp.hlp
;HelpTopic: 106
INCLUDE "WRISTAPP.I"
;
; (1) Program specific constants
;
START     EQU   *
CHANGE_FLAGS    EQU     $92     ; System Flags
SND_POS         EQU     $61
SND_REMAIN      EQU     $62
SND_NOTE        EQU     $63

NOTE_PAUSE      EQU     (TONE_PAUSE/16)
NOTE_BELL       EQU     (TONE_MID_C/16)
;
; (2) System entry point vectors
;
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  rts             ; Called when we are suspended for any reason - WRIST_SUSPEND
nop
nop
L0116:  jmp     CHECKSTATE      ; Called to handle any timers or time events - WRIST_DOTIC
L0119:  jmp     STOPIT          ; Called when the COMM app starts and we have timers pending -
WRIST_INCOMM
L011c:  rts
nop
        nop                     ; Called when the COMM app loads new data - WRIST_NEWDATA

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
rts
L0123:  jmp     HANDLE_STATE0
        db      STATETAB-STATETAB
;
; (3) Program strings
;
```

```
S6_SHIPS:       timex6  “SHIPS”
S6_BELLS:       timex6  “ BELLS”
S8_TOEBES:      Timex   “J.TOEBES”
;
; Here is the pattern for the ships bells.  We want to have a short bell followed by a very short silence
; followed by a longer bell.  We use 3 tics for the short bell, 1 tic for the silence and 6 tics for the
longer
; bell.  The last bell is 7 ticks.
; We then have to byte swap each of these because the BRSET instruction numbers from bottom to top.
;
; The string looks like:
;   111 0 111111 000000 111 0 111111 000000 111 0 111111 000000 111 0 111111 000000
; Taking this into clumps of 4 bytes, we get
;   1110 1111  1100 0000  1110 1111  1100 0000  1110 1111  1100 0000  1110 1111  1100 0000  1111 1110
;
Pattern DB      $F7     ;1110 1111  ; 8 start here
        DB      $03     ;1100 0000
P67     DB      $F7     ;1110 1111  ; 6, 7 start here
        DB      $03     ;1100 0000
P45     DB      $F7     ;1110 1111  ; 4, 5 start here
        DB      $03     ;1100 0000
P23     DB      $F7     ;1110 1111  ; 2, 3 start here
        DB      $03     ;1100 0000
P1      DB      $7F     ;1111 1110  ; 1 starts here
;
; This table indexes where we start playing the tone from
;
STARTS
        DB      (Pattern-Pattern)*8     ; 0 (8 AM,  4PM, Midnight)
        DB      (P1-Pattern)*8          ; 1 (1 AM,  9AM,  5PM)
        DB      (P23-Pattern)*8         ; 2 (2 AM, 10AM,  6PM)
        DB      (P23-Pattern)*8         ; 3 (3 AM, 11AM,  7PM)
        DB      (P45-Pattern)*8         ; 4 (4 AM, NOON,  8PM)
        DB      (P45-Pattern)*8         ; 5 (5 AM,  1PM,  9PM)
        DB      (P67-Pattern)*8         ; 6 (6 AM,  2PM, 10PM)
        DB      (P67-Pattern)*8         ; 7 (7 AM,  3PM, 11PM)
;
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM_LONG,0    ; Initial state
        db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
        db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
        db      EVT_END
;
; (5) State Table 0 Handler
; This is called to process the state events.
; We see ENTER and RESUME events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS             ; Allow us to be suspended
        jsr     CLEARALL                ; Clear the display
        lda     #S6_SHIPS-START         ; Put ‘SHIPS ‘ on the top line
        jsr     PUT6TOP
        lda     #S6_BELLS-START         ; Put ‘ BELLS’ on the second line
        jsr     PUT6MID
        bsr     FORCESTATE              ; Just for fun, check the alarm state
        lda     #S8_TOEBES-START
        jmp     BANNER8
;
; (6) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$C4    ; Bit2 = wristapp wants a call once an hour when it changes (WRIST_DOTIC)
(SET=CALL)
                        ; Bit6 = Uses system rules for button beep decisions (SET=SYSTEM RULES)
                        ; Bit7 = Wristapp has been loaded (SET=LOADED)
        sta     WRISTAPP_FLAGS
        bclr    2,MODE_FLAGS    ; Turn off the hourly chimes
        clr     SND_REMAIN
;
; (7) Determining the current hour
;
CHECKSTATE
brclr   5,CHANGE_FLAGS,NO_HOUR  ; Have we hit the hour mark?
FORCESTATE
        bclr    3,MAIN_FLAGS            ; Make sure we don’t play the system hourly chimes
        jsr     ACQUIRE                 ; Lock so that it doesn’t change under us
        lda     TZ1_HOUR                ; Assume that we are using the first timezone
        jsr     CHECK_TZ                ; See which one we are really using
        bcc     GOT_TZ1                 ; If we were right, just skip on to do the work
        lda     TZ2_HOUR                ; Wrong guess, just load up the second time zone
GOT_TZ1
;
;        12  1  2  3  4  5  6  7  8  9 10 11 12  1  2  3  4  5  6  7  8  9 10 11 12
;        00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18
; deca   FF 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17
; anda   07 00 01 02 03 04 05 06 07 00 01 02 03 04 05 06 07 00 01 02 03 04 05 06 07
        and     #7                      ; Convert the hour to the number of bells
        tax                             ; Save away as an index into the start position table
        bne     NOTEIGHT                ; Is it midnight (or a multiple of 8)
        lda     #8                      ; Yes, so that is 8 bells, not zero
NOTEIGHT
        lsla                            ; Multiple the number of bells by 8 to get the length
        lsla
        lsla
        sta     SND_REMAIN              ; Save away the number of bells left to play
        lda     STARTS,X                ; Point to the pattern of the first bell
        sta     SND_POS
        bset    1,BTNFLAGS              ; Turn on the tic timer
        JMP     RELEASE                 ; And release our lock on the time
;
```

```
; (8) Playing the next note piece
;
NO_HOUR
        lda     SND_REMAIN              ; Do we have any more notes to play?
        bne     DO_SOUND                ; No, skip out
STOPIT
        lda     #TONE_PAUSE             ; End of the line, shut up the sound hardware
        sta     $28
        clr     SND_REMAIN              ; Force us to quit looking at sound
        bclr    1,BTNFLAGS              ; and turn off the tic timer
        rts
DO_SOUND
        deca                            ; Yes, note that we used one up
        sta     SND_REMAIN
        lda     SND_POS                 ; See where we are in the sound
        lsra                            ; Divide by 8 to get the byte pointer
        lsra
        lsra
        tax                             ; and make it an index
        lda     Pattern,X               ; Get the current pattern byte
        sta     SND_NOTE                ; And save it where we can test it
        lda     SND_POS                 ; Get the pointer to where we are in the sound
        inc     SND_POS                 ; Advance to the next byte
        and     #7                      ; and hack off the high bytes to leave the bit index
        lsla                            ; Convert that to a BRSET instruction
        sta     TSTNOTE                 ; And self modify our code so we can play
TSTNOTE brset   0,SND_NOTE,PLAYIT       ; If the note is not set, skip out
        lda     #TONE_PAUSE             ; Not playing, we want to have silence
        brskip2
PLAYIT  lda     #NOTE_BELL              ; Playing, select the bell tone
        sta     $28                     ; And make it play
NO_SOUND
        rts
```

1. Program specific constants - We define the CHANGE_FLAGS because it is not currently in Wristapp.i. This allows us to turn off the system attempts at playing hourly chimes. We also select the tone that we want to play the bells with. This seems to work as the best one to be heard as bells.

2. System entry point vectors - The only interesting thing here is that we use the WRIST_INCOMM entry to disable any bell playing that might have started.

3. Program strings - The pattern and starts tables are used to describe when we will be playing notes and when we will be pausing.

4. State Table - Pretty boring here.

5. State Table 0 Handler - Also amazingly boring. The only interesting thing that we do here is to force the current bells to play when you enter the app.

6. Main initialization routine - Nothing spectacular here, other than the fact that we save 1 byte by falling into the code to determine if we have passed an hour.

7. Determining the current hour - This code looks to see if the hour has changed and if so, it latches in the time based on the selected timezone. It also calculates the number of bells and the length of the sequence necessary to play for that number of bells.

8. Playing the next note piece - The really tricky part here is that we have self-modifying code that generates a BRSET instruction to test the next bit in the currently selected byte. Once we have done so, we load up a tone and stuff it into the hardware.

More Random Numbers and Marquis - PICK6 example

Philip Hudnott <Philip.hudnott@btinternet.com> came up with this idea for a wristapp to pick lottery numbers. Overall, this is pretty simple wristapp to write, but it really showed the need for a decent random number generator. Fortunately, Alan Beale <biljir@pobox.com> provided me with a great MWC (multiply-with-carry) algorithm. Feel free to use the random number generator for other programs, it has some pretty good behavior. Overall, this program has very little changes from the 3BALL example, so getting into it should be pretty easy.

```
;Name: PICK6
;Version: PICK6
;Description: A sample lottery number picker to pick 6 numbers out of a pool of 49 numbers (no duplicates
allowed).
;  To use it, just select it as the current app and it will pick a set of 6 numbers for you.  To get
another set,
;  just press the next button.  This is for amusement only (but if you win anything because of it, I would
welcome
;  anything that you send me).
;
;by John A. Toebes, VIII
;
;HelpFile: watchapp.hlp
;HelpTopic: 100
;****************************************************************************************
;* Copyright (C) 1997 John A. Toebes, VIII                                              *
;* All Rights Reserved                                                                  *
;* This program may not be distributed in any form without the permission of the author *
;*         jtoebes@geocities.com                                                        *
;****************************************************************************************
; (1) Program specific constants
;
            INCLUDE "WRISTAPP.I"
;
; Program specific constants
;
RAND_RANGE      EQU     48      ; This is the number of items to select from (1 to RAND_RANGE+1)
CURRENT_TIC     EQU     $27     ; Current system clock tic (Timer)
RAND_WCL        EQU     $61
RAND_WCH        EQU     $62
RAND_WNL        EQU     $63
RAND_WNH        EQU     $64
THIS_PICK       EQU     $65     ; We can share this with MARQ_POS since we don't do both at the same time
MARQ_POS        EQU     $65
TEMPL           EQU     $66
TEMPH           EQU     $67
START           EQU   *
BASE_TAB        EQU     $FE
;
; (2) System entry point vectors
;
L0110:  jmp     MAIN    ; The main entry point - WRIST_MAIN
L0113:  bclr    1,BTNFLAGS      ; Called when we are suspended for any reason - WRIST_SUSPEND
        rts
```

```
L0116:  jmp     FLASH   ; Called to handle any timers or time events - WRIST_DOTIC
L0119:  bclr    1,BTNFLAGS      ; Called when the COMM app starts and we have timers pending -
WRIST_INCOMM
        rts
L011c:  rts             ; Called when the COMM app loads new data - WRIST_NEWDATA
        nop
        nop

L011f:  lda     STATETAB,X ; The state table get routine - WRIST_GETSTATE
        rts

L0123:  jmp   HANDLE_STATE0
        db    STATETAB-STATETAB
;
; (3) Program strings
;
S6_MARQ timex6  "   +O+   "
S8_TITLE Timex  " PICK-6 "

MARQ_SEL
        DB      S6_MARQ+2-START
        DB      S6_MARQ+3-START
        DB      S6_MARQ+2-START
        DB      S6_MARQ+1-START
        DB      S6_MARQ-START
        DB      S6_MARQ+1-START
;
; (4) State Table
;
STATETAB:
        db      0
        db      EVT_ENTER,TIM2_16TIC,0  ; Initial state
        db      EVT_RESUME,TIM_ONCE,0   ; Resume from a nested app
        db      EVT_DNNEXT,TIM2_16TIC,0 ; Next button
        db      EVT_TIMER2,TIM_ONCE,0   ; Timer
        db      EVT_MODE,TIM_ONCE,$FF   ; Mode button
        db      EVT_END

PICK_VALS       db      0,0,0,0,0,0,0,$FF
;
; (5) This flashes the text on the screen
;
FLASH
        lda     CURRENT_APP     ; See which app is currently running
        cmp     #APP_WRIST      ; Is it us?
        bne     L0113           ; No, so just turn off the tic timer since we don't need it
        ldx     #5
        lda     MARQ_POS
        jsr     INCA_WRAPX
        sta     MARQ_POS
        tax
        lda     MARQ_SEL,X
        jsr     PUT6MID
        ldx     MARQ_POS
        lda     MARQ_SEL,X
        jmp     PUT6TOP
;
; (6) They want us to do it again
;
DOITAGAIN                       ; Tell them we are going to do it again
        clr     MARQ_POS
        bset    1,BTNFLAGS
        jsr     CLEARALL
        jmp     BANNER
;
; (7) State Table 0 Handler
; This is called to process the state events.
; We see ENTER, RESUME, TIMER2 and NEXT events
;
HANDLE_STATE0:
        bset    1,APP_FLAGS             ; Indicate that we can be suspended
        bclr    1,BTNFLAGS
        lda     BTNSTATE
        cmp     #EVT_DNNEXT             ; Did they press the next button?
        beq     DOITAGAIN
        cmp     #EVT_ENTER              ; Or did we start out
        beq     DOITAGAIN
        cmp     #EVT_RESUME
        beq     REFRESH
;
; (8) Select a random answer
;
SHOWIT
        clra
        ldx     #6
CLEARIT
        sta     PICK_VALS-1,X
        decx
        bne     CLEARIT
;
; We want to pick 6 random numbers.  The first needs to be in the range 1 ... RAND_RANGE
; The second should be in the range 1 ... (RAND_RANGE-1)
; The third should be in the range 1 ... (RAND_RANGE-2)
; The fourth should be in the range 1 ... (RAND_RANGE-3)
; The fifth should be in the range 1 ... (RAND_RANGE-4)
; The sixth should be in the range 1 ... (RAND_RANGE-5)
;
        clr     THIS_PICK
ONE_MORE_PICK

REPICK
        jsr     RAND16
        and     #63
        sta     TEMPL
        lda     #RAND_RANGE
        sub     THIS_PICK
        cmp     TEMPL
        blo     REPICK
        lda     TEMPL
        bsr     INSERT_NUM

        inc     THIS_PICK
        lda     THIS_PICK
        cmp     #6
        bne     ONE_MORE_PICK
        bra     REFRESH
;
; (9) Insert a number in the list
;
INSERT_NUM
        inca
        ldx     #(PICK_VALS-1)-BASE_TAB   ; Index so that we can use the short addressing mode
TRY_NEXT
        incx                            ; Advance to the next number
        tst     BASE_TAB,X              ; Is it an empty slot?
        bne     NOT_END                 ; No, try some more
        sta     BASE_TAB,X              ; Yes, just toss it in there
        rts                             ; And return
NOT_END
        cmp     BASE_TAB,X              ; Non-empty slot, are we less than it?
        blo     PUT_HERE                ; Yes, so we go here
        inca                            ; No, Greater than or equal, we need to increment one and try
again
        bra     TRY_NEXT
PUT_HERE
        sta     TEMPL
        lda     BASE_TAB,X
        sta     TEMPH
        lda     TEMPL
        sta     BASE_TAB,X
        lda     TEMPH
        incx
        tsta
        bne     PUT_HERE
        rts
;
; (10) Display the currently selected random numbers
;
REFRESH
        ldx     PICK_VALS
        bsr     GOFMTX
        jsr     PUTTOP12

        ldx     PICK_VALS+1
        bsr     GOFMTX
        jsr     PUTTOP34

        ldx     PICK_VALS+2
        bsr     GOFMTX
        jsr     PUTTOP56

        ldx     PICK_VALS+3
        bsr     GOFMTX
        jsr     PUTMID12

        ldx     PICK_VALS+4
        bsr     GOFMTX
        jsr     PUTMID34

        ldx     PICK_VALS+5
        bsr     GOFMTX
        jsr     PUTMID56

        lda     #ROW_MP23
        sta     DISP_ROW
        bset    COL_MP23,DISP_COL

        lda     #ROW_MP45
        sta     DISP_ROW
        bset    COL_MP45,DISP_COL

        lda     #ROW_TP23
        sta     DISP_ROW
        bset    COL_TP23,DISP_COL

        lda     #ROW_TP45
        sta     DISP_ROW
        bset    COL_TP45,DISP_COL
BANNER
        lda     #S8_TITLE-START ; And show the mode on the bottom
        jmp     BANNER8

GOFMTX  JMP     FMTX
; (11) Here is an excellent random number generator
; it comes courtesy of Alan Beale <biljir@pobox.com%gt;
; The following C code gives a good MWC (multiply-with-carry)
; generator.  This type is generally superior to linear
; congruential generators.  As a bonus, there is no particular advantage to using the high-order
; rather than the low-order bits.
; The algorithm was developed and analyzed by George
; Marsaglia, a very well-known scholar of random number lore.
;
; The code assumes 16 bit shorts and 32 bit longs (hardly surprising).
;
;static unsigned short wn,wc;  /* random number and carry */
;
```

```
;unsigned short rand() {
;    unsigned long temp;
;    temp = 18000*wn + wc;
;    wc = temp >> 16;
;    wn = temp & 0xffff;
;    return wn;
;}
;
;To seed, set wn to anything you like, and wc to anything between 0 and 17999.
;
; Translating this into assembler is
;nHnL*0x4650 + RAND_WCHcL
;
;    unsigned long temp;
;    temp = 18000*wn + wc;
;    wc = temp >> 16;
;    wn = temp & 0xffff;
;    return wn;
;     temp = 0x4650 * n + c
;     temp = 0x4650 * nHnL + cHcL
;     temp = (0x4600 + 0x50) * (nH00 + nL) + cHcL
;     temp = 0x4600*nH00 + 0x4600*nL + 0x50*nH00 + 0x50*nL + cHcL
;     temp = 0x46*nH*0x10000 + 0x46*nL*0x100 + 0x50*nH*0x1000 + 0x50*nL + cHcL
; We construct the 32bit result into tH tL cH cL and then swap the 16 bit values
; once we have no more need of the original numbers in the calculation
;
RAND_MULT       EQU     18000   ; This is for the random number generator
RAND_MULTH      EQU     RAND_MULT/256
RAND_MULTL      EQU     RAND_MULT&255

RAND16
        lda     RAND_WNL        ; A=nL
        ldx     RAND_MULTL      ; X=0x50
        mul                     ; X:A = 0x50*nL
        add     RAND_WCL        ; A=Low(0x50nL)+cL
        sta     RAND_WCL        ; cL=Low(0x50nL)+cL
        txa                     ; A=High(0x50nL)
        adc     RAND_WCH        ; A=High(0x50nL)+cH
        sta     RAND_WCH        ; cH=High(0x50nL)+cH
        clra                    ; A=0
        sta     TEMPH           ; tH=0
        adc     #0              ; A=Carry(0x50nL)+cH
        sta     TEMPL           ; tL=Carry(0x50nL)+cH

        lda     RAND_WNL        ; A=nL
        ldx     RAND_MULTH      ; X=0x46
        bsr     RAND_SUB        ; tL:cH += 0x46*nL  tH=carry(0x46*nL)

        lda     RAND_WNH        ; A=nH
        ldx     RAND_MULTL      ; X=0x50
        bsr     RAND_SUB        ; tL:cH += 0x50*nH  tH=carry(0x50*nH)
        lda     RAND_WNH        ; A=nH
        ldx     RAND_WCL        ; X=cL
        stx     RAND_WNL        ; nL=cL
        ldx     RAND_WCH        ; X=cH
        stx     RAND_WNH        ; hH=cH
        ldx     RAND_MULTH      ; X=0x46
        mul                     ; X:A=0x46*nH
        add     TEMPL           ; A=Low(0x46*nH)+tL
        sta     RAND_WCL        ; nL=Low(0x46*nH)+tL
        txa                     ; A=High(0x46*nH)
        adc     TEMPH           ; A=High(0x46*nH)+tH
        sta     RAND_WCH        ; nH=High(0x46*nH)+tH
        rts

RAND_SUB
        mul                     ; Compute the values
        add     RAND_WCH        ; A=LOW(result)+cH
        sta     RAND_WCH        ; cH=Low(result)+cH
        txa                     ; X=High(result)
        adc     TEMPL           ; X=High(result)+tL+Carry(low(result)+cH)
        sta     TEMPL           ; tL=High(result)+tL+Carry(low(result)+cH)
        clra                    ; A=0
        adc     TEMPH           ; A=carry(High(result)+tL+Carry(low(result)+cH))+tH
        sta     TEMPH           ; tH=carry(High(result)+tL+Carry(low(result)+cH))+tH
        rts
;
; (12) This is the main initialization routine which is called when we first get the app into memory
;
MAIN:
        lda     #$c0            ; We want button beeps and to indicate that we have been loaded
        sta     WRISTAPP_FLAGS
        lda     CURRENT_TIC
        sta     RAND_WNL
        sta     RAND_WNH
        sta     RAND_WCL
        and     #$3f
        sta     RAND_WCH
        rts
```

1. Program specific constants - We have several variables - RAND_WCL, RAND_WCH, RAND_WNL and RAND_WNH which we use for the random number routine. CURRENT_TIC is what is set by the system when it reads the clock to keep the watch time up to date. We use it once to provide a seed for the random number generator. Note that we are overlapping the use of THIS_PICK and MARQ_POS to save one byte of low ram.
2. System entry point vectors - identical to the 3BALL example, This one gets to be a little fun. Notice for the WRIST_SUSPEND and WRIST_INCOMM routines that we don't have a JMP instruction, but instead put the actual code in line. This saves use a couple of bytes.
3. Program strings - We are pretty frugal here in reusing blanks at the end of the string very liberally. Also note the S6_MARQ string which has blanks at the start and end so that it can shuffle left and right on the display but always have blanks visible. The MARQ_SEL and MSG_SEL tables are simply offsets that allow us to select the message with a simple load instruction instead of having to calculate the offset.
4. State Table - This is pretty vanilla here except for the fact that we have a very long time interval after the DNNEXT and ENTER events. It is during this time that the Marquis runs. We could make it even longer,

but this seems to be a good compromise between seeing something happen and actually getting a result in a reasonable time.

5. State Table 0 Handler - Extremely simple, there are only four events that we want to see and this is the typical test and branch one. The only unique thing here is that we turn off the Marquis timer as soon as we get any event.
6. This flashes the text on the screen - This is the cheap way to do a Marquis. Just have a string wider than the display and change the offset from the start at which you start to display. For this one, there are only 6 states and we select the starting offset from the table based on our current cycle. Note that this routine is called by the TIC timer which is enabled when they want a new random number. Eventually the timer for the main event will run out and they will simply stop calling us.
7. They want us to do it again - Whenever we want to do a new random number, we just start the Marquis tic timer and set up the display.
8. Select a random answer - This is really the meat of this wristapp. We need to pick 6 random numbers and sort them. Fortunately, we can take advantage of the sorting as part of our random number selection.
9. Insert a number in the list - Given a random number, add it to the list of random numbers in sorted order. Essentially, we start at the beginning of the list and go until we either find a slot where we need to insert the number in order or we hit the end of the list. If we hit the end of the list, we store the number there and return. Otherwise we insert the number at the appropriate spot. One additional thing that we do is increment the number by 1 for each entry in the that is less than it. It makes sense, but you need to think about why this works.
10. Display the currently selected random numbers - Given the 6 random numbers, we just put them on the display separated by periods. Note the series of BSR instructions to the GOFMTX label. Since there were 6 calls to it, we were about to reduce the 6 3-byte instructions to 6 2-byte instructions plus one 3-byte instruction to do the call for a savings of 3 bytes.
11. Here is a random number generator - This is great random number generator that you might want to grab for any other code that you might write.
12. This is the main initialization routine which is called when we first get the app into memory - Very boring stuff here, but we do take a moment to initialize the random number seed with the current tic count just to make it a little more variable.

Index