THE INHERENT INSECURITY OF DATA OVER MOBITEX WIRELESS PACKET DATA NETWORKS

TABLE OF CONTENTS

1. Introduction

The Mobitex wireless data network standard was originally developed in Sweden in the 1980's and successfully marketed by Ericsson. Over the years Mobitex networks have sprouted up in numerous countries around the globe as Ericsson's world wide web page (http://www.ericcson.com) points out. So no matter where you are reading this there might be Mobitex network in operation near you. If so, this posting and attached program gives you all the information you need to intercept and decode whatever information is flowing over your local system. The required equipment is easily availble - you will need a scanner radio with discriminator output, a PC compatible computer, and a simple Op-amp based audio clipping circuit of the Hamcomm or PD203 Pocsag decoder type.

The reader will find that a lot of information in this posting is geared towards one specific Mobitex network operated by RAM Mobile Data in the United States. Nonetheless, the included program has been written with flexibility in mind - with the proper configuration options set it should work with most any Mobitex system.

Finally, one may ask whether this project is really worth the effort. The answer to that question probably depends on what sort of Mobitex systems are active in reader's vicinity. And one sure way of finding that out is by getting the described decoding system up and running. At that point the authors suppose it becomes a matter of personal taste - some people might find it a most satisfyingly orgasmic experience to watch appliance repair personnel being dispatched around their town while others move on to better things. For those contemplating monitoring the RAM network: it should be mentioned that this network uses a cellular architecture which means that traffic in your city will probably be divided over a number of channels reducing the number of messages you can intercept. The authors will say though

that RAM's network does have plain ASCII messages whizzing around, one might just have to be a little patient.

****
2. The Mobitex Protocol
****

Basically we'll let an excerpt from RAM's whitepaper available on the world wide web do all the work.

from http://www.micron.net/~rmd/press/white_paper/security.html

> RADIO SIGNALING PROTOCOL (ROSI)
> ———————————————————————————————————————
>
> The MOBITEX network uses a Time Division Multiple Access (TDMA) method
> with a modified Slotted Aloha channel access algorithm. This unique
> channel access algorithm provides an efficient channel utilization.
> Data is sent over the airlink in short bursts at 8000 bps using
> Gaussian Minimum Shift Keying (GMSK) modulation, encoded and
> interleaved for error correction, and then scrambled.
>
> Messages in the MOBITEX system are made up of packets (called MPAKs)
> at the network layer (OSI layer 3). In the radio modem, these are
> encapsulated for transmission over the radio link by protocols at the
> data link (OSI 2) and physical (OSI 1) layers. The data link layer of
> the radio modem performs all the required base band signal processing
> operations. The PMAKs are broken up into radio frames by the radio
> modem. A frame header of 56 bits is generated and placed in front of
> each radio frame. Data blocks consist of 20 bytes, each containing 8
> bits of data followed by four bits generated with a (12,8) Hamming
> code. A radio frame has up to 20 data blocks. In order to protect
> against burst error, interleaving in performed. The interleaving
> process can be viewed by imagining the data arranged as rows in a 12
> bits wide by 20 bits long matrix. The data is then sent column wise.
> Up to this point, the Hamming code corrects single bit errors per
> byte, and the interleaved code allows a burst of 20 errors to be
> corrected. The data following the frame head is then scrambled.
> Scrambling provides a mechanism to achieve DC voltage balance by
> eliminating long sequences of ones and zeros in the data stream. At
> this point, we have completed the necessary digital operations and are
> ready for RF processing. The modem then performs the GMSK modulation
> before applying the signal to the radio transceiver and onto the
> appropriate radio channel. The receive operation is the exact reverse
> of the transmit operation. A block diagram of the base band processing
> operations is shown in Figure 4
>
> Receiving and transmitting data over the airlink in the MOBITEX
> network is a very complicated process. It is a difficult task for
> radio modem manufacturers, not to mention much less sophisticated

> hackers trying to break into the network. It may be more cost
> effective to break into the user's computer system, rather than the
> MOBITEX network since the technical expertise needed to break into the
> air interface is so extensive.
>

Now just a brief explanation / amplification of some of the terms:

Aloha channel access - when the mobile transmits a message it expects an acknowledgement from the base station. If it doesn't get it the mobile assumes it transmitted at the same time as another mobile so it waits a short, random period and tries again.

GMSK - This is basically the same thing as direct frequency shift keying except the waveform is filtered to take all the sharp edges off so it can fit in a narrower channel. This makes it more susceptible to noise, but hey - you can't have everything.

BIT SCRAMBLING - like RAM says, it is done purely for technical reasons, not to hide the contents of the data (but try explaining that to a U.S. court after you get arrested for violating ECPA laws). The scrambler is just a regular 9 stage shift register pseudo-random white noise generator with feedback taps on the fifth and ninth stages initialized at the start of the first data block. What if you didn't know this? The interested reader will verify that if a correlation were performed on data stream with two shifted versions of itself (just perform a normal correlation twice in a row) you can easily see that when one stream is shifted by 5 and the other by 9 bits that something special is happening. The tranmsitted data is actually correlated enough for you to directly pick out the correct shift register taps! The interested reader might also want to verify that a regular correlation using only the bit stream and a single shifted version of itself is absolutely useless. Those working on the key recovery problem for certain versions of Motorola's digital voice privacy (DVP) encryption using similar shift register techniques might want to proceed along those lines (of course there are a hell of a lot more taps to worry about).

FRAME HEADER -
   For all Mobitex systems the first two bytes are bit sync:
           CCCC for the base and 3333 for the mobiles.
   The next two bytes are a system specific frame sync.
           For RAM this will be B433.
   The next two bytes are control bytes (again system specific):
           FOR ram the first part gives system ID, the latter part is a
           status. When Bits 1 and 2 of the second control byte are zero
           then data blocks will follow the frame header.
   The last byte provides FEC for the two control bytes.

   The program detects the Frame header by matching up the bit sync, the frame sync (optional), and checking the FEC coding on the two control

bytes. When the frame header is detected it will either assume that a data block is following the frame header (non RAM mode) or it checks the control byte to see if a data block follows the frame header (RAM mode). The frame sync is not necessarily followed by any data.

DATA BLOCK - the program processes the data block as follows:
undo the bit scrambling (if that option is set)
uninterleave the data
Run through the forward error correction (FEC) routine while keeping track of the number of errors.
Calculate the CRC on this data block and compare with the received CRC.
If the calculated CRC does not match the transmitted CRC and there were a large number of FEC errors the programs decides that a data block was actually not transmitted and goes back to hunting for the frame header.
If a good data block was received it will go on and try to receive another data block.

Basically then, the program cycles back and forth between hunting for the frame sync and printing out DATA blocks. The output format of the data blocks will look something like:

6F6C6F6775653A2049424D204D6F62696C65A73B ologue: IBM Mobile§;

or if the calculated CRC was bad it would have looked like:

6E6C6F6775653A2049424D204D6F62696C65A73B BAD CRC nlogue: IBM Mobile§;

The last two bytes are the CRC; the first 18 bytes are the data. Whenever you see these types of blocks appearing on the screen then you can rest assured that you do indeed have the program configured properly. Keep in mind that your system might have very little actual traffic. Also, status message will appear as single data blocks (at least for the RAM network).

If you the reader must know more then take a look a various semiconductor data sheets (example : look at the MXCOM's 909 Mobitex interface chip for a data sheet giving a lot of information on the mobitex protocol - http://www.mxcom.com/mx909Api.htm). Also, more details on the RAM system can be found in the following article:

M. Mobeen Khan, "Wireless dta over RAM's Mobitex network," Proceedings of the SPIE, Vol. 2601, pp. 48 - 53, 1995.

Of most interest is section 3 giving information on the network protocols. This would help one figure out the raw data blocks displayed by this program. The basic upshot - the first data block of a packet has the following format:

BYTES 1-3 : Destination Mobitex Acess Number (MAN) - tells which

```
                mobile is to receive this packet
    BYTES 4    : Frame ID
    BYTE  5    : Sequence number
    BYTE  6    : Number of data blocks in the packet

    The above 6 bytes are part of the Mobitex standard (data link
    layer). The next are specific to the RAM network layer
    implementation.

    BYTES 7-9   : Sender's MAN
    BYTES 10-12 : Addressee MAN
    BYTE  13    : State Flag
    BYTE  14-18 : Packet ID

    BYTE  19-20 : CRC bytes - we're back to the Mobitex standard
```

The state flag can be used to send any one of 256 status messages to a specific terminal (this is application dependent). So don't be suprised to see a lot of single data blocks coming in over the air. The next data blocks (if any) are user data which can be virutally any kind of protocol. The included program does not take advantage of any of this fancy stuff - it simply displays the data blocks sent over the physical layer.

For additional "security" each device that works on the RAM network has its own serial number in addition its mobitex access number. This serial number is verified every time the mobile logs on to RAM. No doubt readers will recognize that this is exactly the same process used to provide security for cellular phones in north America. One can only conclude that the RAM Mobile Data network is similarly waiting to be abused. The attached program could potentially be used to gobble up lots of MANs and their corresponding serial numbers as part of a cloning operation. The only reason that hasn't happened yet is that cloning cellular phones is a far more lucrative proposition. However, please don't misunderstand us - we're not trying to advocate or promote any sort of illegal activities.

Conclusions: Mobitex is truly the greatest contribution to the telecommunications industry from the Nordic countries. At this point the authors would be remiss if they failed to mention the greatest contribution to humanity from this part of the globe - mass quantities of cheap and legal pornography! What telecom nerd could ask for anything more?

****

3. How to identify RAM / Mobitex frequencies

****

In the United States, RAM mobile data has built an extensive Mobitex network covering most of the larger cities. To verify RAM's

coverage in your area take a look at the following link:

http://www.ram-wireless.com/coverage/coverage_main.html

Once you have verified that you are indeed in the RAM coverage area you will want to find the right frequencies. In each of these areas RAM will have obtained at least one block of 10 individual frequencies somewhere in the 935-940MHz (Base) / 896-901MHz (Portable) band (with 12.5KHz channel spacing). The specific frequencies allocated vary from area to area. If you don't have access to a frequency database one can find the RAM's frequencies by scanning for channels with the following characteristics. The transmitter is always on. When the channel is idle there will be a background information burst transmitted roughly every half second or second. The best description for how a packet sounds would be a brief burst of white noise. Between packets the carrier will be unmodulated and you won't hear a thing. This means you want to find those channels that sound like "Chuff... Chuff... Chuff..." I don't know whether other Mobitex networks sound similar - after all, one needn't necessarily keep the transmitter on all the time, one might be running at a different baud rate, and if bit scrambling is not used the system might sound less like noise.

The program options section gives a suggested procedure to follow for trying to receive unknown Mobitex systems.

As an aside, anyone who has the ability to monitor Motorola mobile data terminals using the MDC4800 protocol (using a different program available on the internet) can monitor RAM's main competitor ARDIS. Their coverage maps and more information can be found at:

http://www.ardis.com/ardis_hp/mapintro.htm

****

4. Stupid stuff

****

The program itself is absolutely free, not shareware. As such, the authors do not feel obligated to provide any support or updates beyond the information included in this text. Any attempts to personally contact the authors are doomed to failure since they have taken steps to remain anonymous.

There are no restrictions on redistribution of this text file or the program. Mangle it, rewrite it, sell it (like you're going to get any money for this in the first place), whatever... see if the authors care - that's why there giving this away for free.

If anyone has more information to add please post! Let us know of any major problems you run into. Please post to the rec.radio.scanner newsgroup - the authors are so cowardly that they do not want to admit to having regular internet access (which explains in part why this is

being posted from a public terminal). Also, let everyone know if you find any active Mobitex systems other than RAM Mobile Data's and what they are used for.

Anyone archiving this information or putting it on some world wide web page will probably want to delete the following paragraph. One of the authors absolutely insisted this paragraph be included although the others would have preferred not to.

The authors exhort anyone reading this not be a parasitic internet douche sponge requesting and soaking up information for financial and egotistical reasons while giving nothing of value in return. As an illustration, an author profile of jvpoll@dallas.net taken from www.dejanews.com uncovers his step by step progress in the past year in attempting to build a device to track and follow trunked radio systems - pleadings on the sci.crypt newsgroup for help on decoding trunked radio control channel protocols, discussions on the sci.electronics.design newsgroup on how to protect his "proprietary" code and help on hooking up a LCD display, the incessant stream of advertising and ego building on the alt and rec.radio.scanner newsgroups, in addition to postings to various alt.recovery.* and alt.romance newsgroups. An author profile from www.dejanews.com only scratches the surface - it won't reveal things like membership in the North American Man Boy Love Association (apparently child molestors try to avoid public exposure) or how thick the layers of dried up semen crusts are on Jim's trousers and commercial products (potential customers are advised to delay ordering from Jim until male impotence finishes taking its terrible toll).

****

5. Humor Section

****

For your entertainment this section gives excerpts from what two leading communication companies (RAM Mobile Data and Bellsouth) have to say about Mobitex data security on their web pages. If you don't believe these are real check out the http links for yourself.

RAM Mobile Data -

from http://www.micron.net/~rmd/press/white_paper/security.html

> The combination of digital technology and packet data switching has
> produced a high degree of inherent security to safeguard the privacy
> of users' data. The additional use of sophisticated protocols and
> unique radio modem designs make it extremely difficult to tap into
> the MOBITEX networks.

> Receiving and transmitting data over the airlink in the MOBITEX

> network is a very complicated process. It is a difficult task for
> radio modem manufacturers, not to mention much less sophisticated
> hackers trying to break into the network. It may be more cost
> effective to break into the user's computer system, rather than the
> MOBITEX network since the technical expertise needed to break into
> the air interface is so extensive.

> There are many inherent security measures built into MOBITEX and a
> wide selection of network features capable of enhancing data
> security. Unless one is transmitting very sensitive data, one need
> not be apprehensive about data security on the MOBITEX networks.

Bellsouth - from http://www.data-mobile.com/unshocked/security.html

> Mobitex protocol provides a high level of security. Data
> transmissions over a wireless packet switched network are much more
> difficult to capture than voice transmissions over a cellular voice
> network. Unlike conversations in the cellular environment, which are
> continuous and easily monitored by unsophisticated eavesdroppers,
> messages in a packet switched environment are sent in bursts.
> "Reading" such messages is only possible if the RF interface can be
> descrambled, a process requiring a level of personnel skill and
> software sophistication that is prohibitive. In addition, Mobitex is
> compatible with customer selected security packages, thus enabling
> the user to choose additional security for select messages.

Conclusions: BULLSHIT. Especially when Mobitex is a public, freely available standard. The most challenging part of this project was actually getting the Mobitex documentaion, everything else was a programming excercise. Not that proprietary protocols would have been more difficult to figure out - just ask Motorola!

Obviously the users of Mobitex networks have been lied to and raped by large corporate entities. The only real way to achieve data security over a Mobitex network is using real honest-to-god encryption. This decision is left up to each customer. The reader is invited to make an inspired guess as to how many customers actually choose to use real encryption after being told that it is virtually impossible for anyone to receive data transmitted over the radio link. The authors of this posting sincerely hopes that all those developers of portable terminals using RAM's network for credit card verification didn't listen too closely to what RAM was telling them.

The authors urge everyone to contact these companies and point out to them the error of their ways. When doing so be sure to emphasize you're reading data off a RAM's network with an el-cheapo scanner, a few dollars worth of parts from Radio Shack, and some really shitty code floating around the internet. It can't hurt to lie and tell them you're

a HACKER (god forbid!) going by a handle such as 'Brunhilde' and that you're using your skills to collect dozens of credit card numbers off of RAM's network each day. Maybe then they'll decide (along with Morotola) to do something to secure their mobile data terminal systems such as bribing enough Congress-creatures to pass a law mandating 20 year prison sentences for anyone who has ever bought an op-amp from Radio-Shack (and now you know why Radio-Shack maintains a computer database of all of its customers).

****
6. The computer to scanner interface
****

The interface described here will allow you to monitor not only Mobitex systems, but with different freely available software packages you can also monitor POCSAG paging signals (PD203 package), Motorola MDC4800 mobile data terminal transmissions (just ask on this newsgroup) decode Motorola trunked radio control channel information (see next blurb about a program called trunker.exe), and also shortwave RTTY and fax transmissions (HAMCOM and JVFAX software packages). Not too bad for a few dollars worth of parts!

------ IRRELEVANT NON-MOBITEX INFORMATION WARNING!!! ------

Some semi-new Motorola trunked radio control channel information for those poor souls palying around with trunker.exe: Those who might want to monitor 800MHz systems operating with some channels offset every 12.5KHz might want to look at my current best guess at the algorithm for converting a frequency assignment code into an actual frequency:

1. If the code is < 0x2D0 then
   Frequency = 851.0125 + (0.25 * code)

2. For the remaining codes: if the code < 0x2f8 then
   Frequency = 848.0000 + (0.25 * code)

3. If 0x32f < code < 0x340:
   Frequency = 846.6250 + (0.025 * code)

4. If the code is > 0x3C0:
   Frequency = 843.4000 + (0.025 * code)

I'm pretty sure 0x2f8, 0x340, and 0x3C0 are not frequency assigments - they define (within a spot or two) boundaries between blocks of information and frequency assignments. I've only been able to test this algorithm out on one system that used frequencies falling under cases 2,3, and 4 (anyone from the rural North Alabama or middle Tennessee areas will sympthasize with my plight) so there are no guarrantees that it works properly for every single frequency in your area. Any corrections to this information are welcome and thanks to whoever posted

the right formula for 900MHz systems. And if you do post a message saying it's all screwed up include exactly which frequencies in your area aren't getting decoded correctly and maybe what this algorithm thought they were.

For those who want to send information to your scanner and have trouble configuring the serial port you will definitely want to remove the following line from the set8250 routine:

```
  outportb (0x03fb, 0x00);     /*  set IER on 0x03f9 */
```

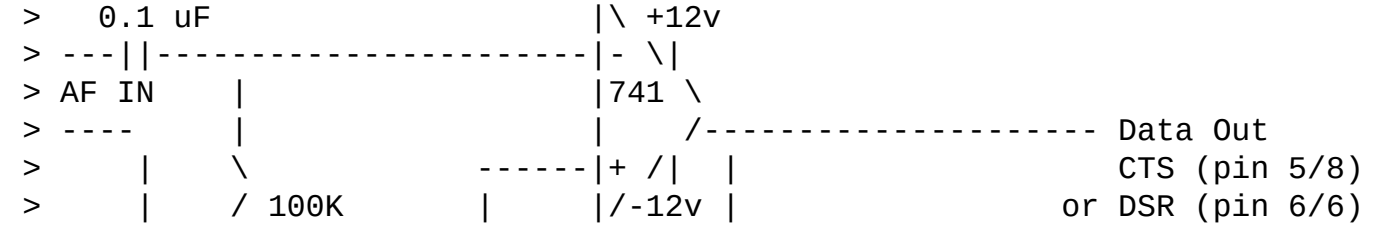
What this line did was to effectively undo the serial port parameter setup from just a few lines earlier.

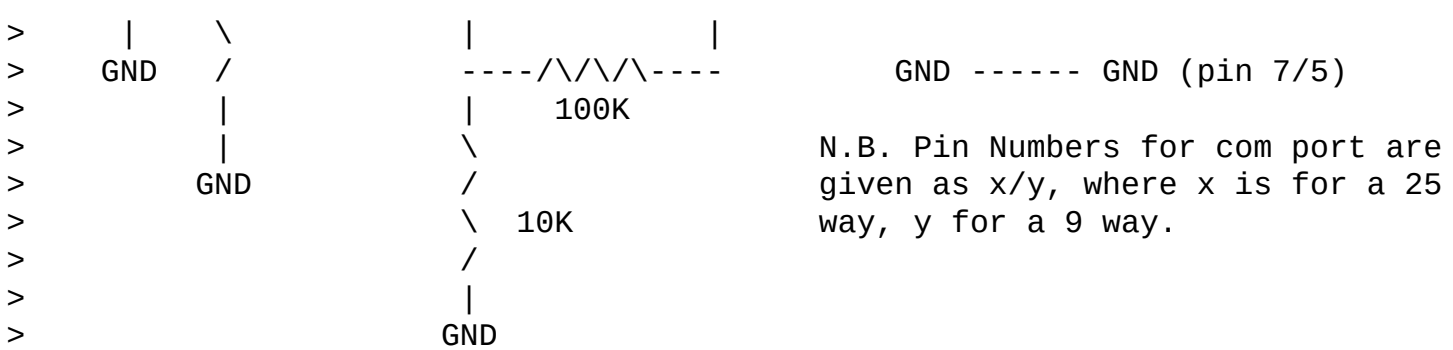
Finally, those of you messing around with ways to tell your computer when to switch back to the control channel might want to just detect the disconnect sequence (11001010 sent at 300 baud). This works even using just your present interface (no audio is being transmitted at that time) without an additional low-pass filter section feeding into your computer just for sub-audible information. It's incredibly cheesy but you'll find it actually works.

---------- Back to Mobitex stuff ----------

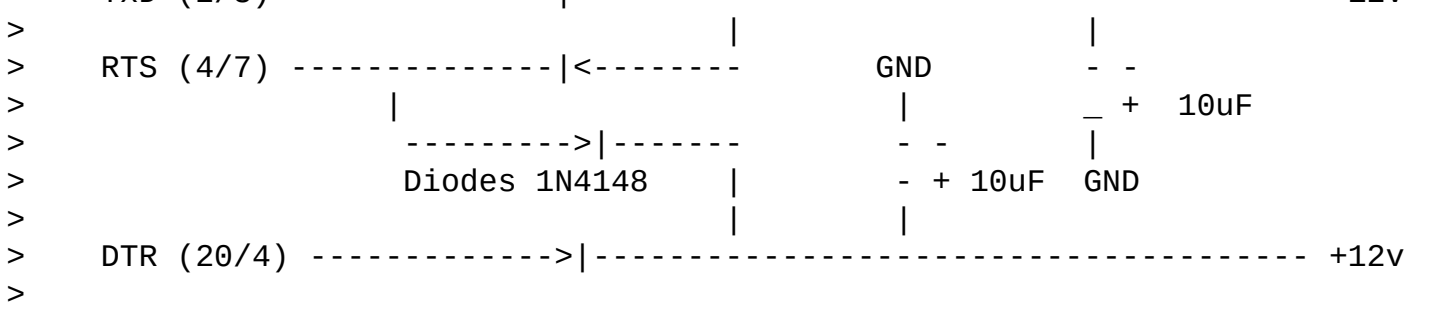
Now for the really fun part - your scanner will need to be modified to allow you to tap off directly from the discriminator output. If you wait until the signal has gone through the radio's internal audio filtering the waveform will likely be way too heavily distorted to be decoded. This is exactly the same problem that our friends who like to decode pager transmissions run into - some of them have claimed they can only decode 512 baud pager messages using the earphone output of their scanner. These transmissions are 8000 baud so you absolutely must use the discriminator output. If you don't know where to begin modifying your scanner you might want to ask those who monitor pagers how to get the discriminator output for your particular radio.

Those of you who have already built an interface for decoding pager messages should be able to use that interface with some possibly minor modifications. For those people starting from scratch - you might want to check out packages intended for pager decoding such as PD203 and the interfaces they describe. The following excerpt gives an example of a decoder that is a good place to start experimenting (lifted out of the PD203 POCSAG pager decoder shareware documentation):





> The above circuit is a Schmitt Trigger, having thresholds of about +/- 1v.
> If such a large threshold is not required, eg for a discriminator output,
> then the level of positive feedback may be reduced by either reducing the
> value of the 10K resistor or by increasing the value of the 100K feedback
> resistor.
>
> The +/- 12v for the op-amp can be derived from unused signals on the COM
> port (gives more like +/- 10v but works fine !):-
>



DO NOT use the RI (ring indicator) as an input to the computer.

The authors strongly suggest tying the non-inverting (+) input of the op-amp to ground since you are working directly with the discriminator output and don't need a Schmitt trigger.

As an example, the authors are currently using the using the above circuit with the following characteristics:

1. The non-inverting input was tied straight to ground
2. The capacitance value was reduced to 0.02 microfarads.
3. The 100K resistor at the capacitor was raised to 2 MegaOhms
4. The op-amp used was an LF411

These changes were done primarily to reduce the loading on the discriminator output circuitry of the scanners used (one presumes the intial audio stages have high impedances). The capacitance had been reduced to 0.02uF to avoid the following problems while monitoring Motorola Mobile data terminals - when a transmitter kicked on it took such a long time for the DC level discriminator output to re-establish itself that the very first message was often cut off. In fact, when

monitoring Motorola MDC4800 mobile data terminals the authors feel it prudent to revert back to a 100Kohm resistor. The moral of this story is play around and use your oscillscope to verify the interface is working exactly right!

Interface checking - use of the PD203 POCSAG pager decoding program is a good test. Additionally, the included program will tell you whether it thinks the interface is working during the startup process. This test only means that the input lines on the serial port are successfully changing states. It doesn't guarantee that your interface or scanner is suitable for 8000 baud operation.

Keep in mind that depending on your scanner and interface you may need to set the /BI:1 option in the program to invert the polarity of the incoming data.

A few words about your scanner - the receive bandwidth must be wide enough to pass the 12.5KHz wide Mobitex channel without problems. I doubt this will a problem for any scanner since they are built to receive 25KHz channels. However, if you have a narrow FM mode setting for 12.5 KHz channels then the filtering might produce too much distortion. So you might want to start experimenting with the regular setting that you use to monitor 25 KHz channels.

****

7. A Few words about the program

****

Remember - don't run this under Windows. DO NOT use control-c unless you enjoy rebooting your computer on a regular basis (but you're probably used to doing that with Microsoft products). To exit this program hit any key on the keyboard. This program does not accept any sort of keyboard input - all configurations are done with the command line arguements listed below. An example of this would be (assuming you called your program executable is called "mobitex.exe"):

```
mobitex /v:1 /cfs:0 /rn:0 /com:2
```

By default the program stores all received data in a file starting with MOBYDA. Each time the program is run it picks a new filename so that your previously captured data will not be overwritten. The program defaults are set for operation with RAM's mobitex network (the only parameter you might need to set is the bit inversion option which will depend on your particular scanner and interface). The authors suggest using a one line batch file with all the configuration information in it for each type of system you will want to monitor.

By default, the program assumes you will be listening to a base station, not a portable computer.

Finally, are you worried that your computer is too slow to keep up

with this mess? The very last number that the program prints to the screen gives approximately the maximum value of how full the receive buffer got. On the authors' systems running fast Pentiums this value rarely exceeds 100 out of the full 15000 word buffer length.

- -
Summary and explanation of program command line arguement options

VERBOSE MODE: when enabled the program will show all received frame sync and control bytes as they are received. You will want to set this mode if you are examing an unknown system and want to verify that this is indeed a Mobitex system.
/V:1 - print received system frame sync and control bytes
/V:0 - don't show the above (DEFAULT)

CHECK FRAME SYNC: If the frame sync option is correctly set for the system you are monitoring then enable this option to better allow the program to decode incoming data. If you don't know the frame sync disable this option and enable the /V option to find out what it is.
/CFS:1 - check system specific frame sync field (DEFAULT).
/CFS:0 - Don't check it - useful for unknown systems.

FRAME SYNC: Each system is allowed to have its own frame sync sequence. This option allows you to set it to whatever your particular system is using. Once the proper Frame sync is specified you can enable the CFS option.
/FS:XXXX - set system specific frame sync to HEX XXXX (DEFAULT B433)

BIT SYNC: The Mobitex standard specifies that this should be CCCC for transmissions from a base station. The only reason that this is a user option is because if you want to listen to the mobile transmissions you must change this setting to 3333. These are the only two settings that you should ever need.
/SY:YYYY - set bit sync to HEX YYYY (Default for base stations CCCC)

BAUD RATE: You can tell the program to run at virtually any baud rate you want. If you don't know the baud rate of an unknown system you probably should look at your received waveform on an oscilloscope.
/BR:Z - Z is the Mobitex system's baud rate (Default = 8000).

BIT SCRAMBLING: This option allows you to turn the bit scrambling off if needed although the authors guess that most systems will be using it.
/BS:1 - System uses bit scrambling (DEFAULT)
/BS:0 - system does not use bit scrambling

RECEIVE CLOCK ADJUSTEMENTS: The crystal controlling your computer's serial port might be slightly different from the author's. The current routine will attempt to make up for this variation (within +/- 5 ns) of the 840ns clock.
/CA:1 - program attempts to fine tune receive clock (DEFAULT)
/CA:0 - program does not attemp to fine tune receive clock.

BIT INVERSION: Your scanner and interface combination may invert the
  incoming data. If so this option must be set.
  /BI:1   - Inverts incoming raw data.
  /BI:0   - don't invert incoming raw data (DEFAULT).

RAM NETWORK mode: If set, the program decides that a sync and control byte
  sequence is followed by data blocks if the second control byte has
  bits 1 and 2 zero. If not set then the program checks to see if a data
  block follows EVERY sync/control byte sequence. This probably should
  be set to zero for non-RAM systems.
  /RN:1   - RAM only: control bytes determine if data blocks follow sync.
  /RN:0   - always try to decode data blocks regardless of control bytes.

OUTPUT FILE: If you don't want data written to an output file (with a
  unique filename) disable this option. This might be useful for
  troubleshooting if you suspect disk access is causing problems.
  /OF:1   - Write data to output file (DEFAULT).
  /OF:0   - Don't write output data file.

COM PORT: Allows you to set which com port you want to use with your
  interface.
  /COM:z   - set z = 1,2,3,4 to set com port you want to use.

- -
Suggested procedure for an unknown Mobitex system:

1. Find out the correct baud rate. Best bet - use an oscilloscope.

2. Start out with following options set: /RN:0 /CFS:0 /v:1

3. If you don't known whether you need to invert the incoming data the
authors suggest the following procedure: try both /BI:0 and /BI:1
options. The correct setting will probably be the one which gets good
control bytes (you don't see the "BAD Control Byte FEC" message as
often).

4. You want to look at those lines with good control bytes and indentify
the system specific Frame sync. From now on, when you want to monitor
this system use the /FS: and /CFS:1 options to tell the program to
actively check for this frame sync. If things are working reasonably
well at this point you should only good control bytes appearing on the
screen every time you hear some data coming over the channel.

5. At this point the only option you have left to play with is the /BS:
option. You will known if the /BS: option is correctly set when you see
valid data blocks appearing on the screen - they will look something
like :

6F6C6F6775653A2049424D204D6F62696C65A73B   ologue: IBM Mobile§;

If you see any blocks of this form then all parameters are correctly set - such blocks had to go through several hurdles involving forward error correction and a CRC check and would have not been displayed if program parameters have been incorrectly set.

6. Sit back and snarf the data. In order not to see the same frame sync and control bytes from appearing over and over you now probably will want to set the /V:0 option. Keep in mind that very little actual data may be flowing over the network you are monitoring. Because of this the authors have been known to leave their computers and scanners monitoring a specific Mobitex system all day while they are off at work.

- -

As far as compiling this - save the latter portion of this posting (the program listing) and feed it to a C compiler. Pretty much any C compiler from Borland should work. If you (Heaven Forbid) use a Microsoft C compiler you might need to rename some calls such as outport. Follow any special instructions for programs using their own interrupt service routines (my compiler package recommends turning the "Use register variables", and "stack warning" compile options off). It also does not want anything whatsoever to do with Windows. Please don't even think about running this program under Windows.

****
8. Finally - the program listing itself
****

---------- C u t H e r e ! ! ! ----------

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <string.h>

/*****************************************************************/
/*                  GLOBAL VARIABLE DECLARATIONS                 */
/*****************************************************************/
int lc=0;
char ob[1000];  /* output buffer for packet before being sent to screen */
int obp=0;      /* pointer to current position in array ob              */
double dtt, odtt,ul,ll; /* variables used in clock tracking routine     */

FILE *out;      /* pointer to output file                               */

/* configuration parameter declaration to the defaults for RAM network  */
static int verbose = 0; /* verbose mode parameter */
static int cfs = 1;     /* system specific frame sync checking status   */
static int frsync=0xB433;       /* system specific frame sync           */
```

```
static int btsync=0xCCCC;       /* bit sync                              */
static double brate = 8000.0;   /* Mobitex system baud rate              */
static int bitscr = 1;          /* bit scrambling in use ?               */
static int clocka = 1;          /* fine tune receive clock               */
static int bitinv = 0;          /* bit inversion                         */
static int ramnet = 1;          /* ramnet flag - 1 means it's ram        */
static int outfil = 1;          /* file output toggle                    */
static int comport = 0x3f8;     /* serial port base address; set in main*/
static int tempo=0;             /* maybe I should get a new compiler     */

/* data buffer for raw data coming in over serial port                  */
static unsigned int buflen= 15000;      /* length of data buffer        */
static volatile unsigned int cpstn = 0; /* current position in buffer   */
static unsigned int fdata[15001] ;      /* frequency data array         */

void interrupt (*oldfuncc) (); /* vector to old com port interrupt      */


/*********************************************************************/
/*                  SERIAL PORT INTERRUPT HANDLER                    */
/*********************************************************************/
/*                                this is serial com port interrupt    */
/* we assume here that it only gets called when one of the status      */
/* lines on the serial port changes (that's all you have hooked up).   */
/* All this handler does is read the system timer (which increments    */
/* every 840 nanoseconds) and stores it in the fdata array. The MSB    */
/* is set to indicate whether the status line is zero. In this way     */
/* the fdata array is continuously updated with the length and the     */
/* length and polarity of each data pulse for further processing by    */
/* the main program.                                                   */
void interrupt com1int()
{
  static unsigned int d1,d2,ltick,tick,dtick;

  /* the system timer is a 16 bit counter whose value counts down     */
  /* from 65535 to zero and repeats ad nauseum. For those who really  */
  /* care, every time the count reaches zero the system timer         */
  /* interrupt is called (remember that thing that gets called every  */
  /* 55 milliseconds and does housekeeping such as checking the       */
  /* keyboard).                                                       */
  outportb (0x43, 0x00);       /* latch counter until we read it      */
  d1 = inportb (0x40);         /* get low count                       */
  d2 = inportb (0x40);         /* get high count                      */

  /* get difference between current, last counter reading            */
  tick  = (d2 << 8) + d1;
  dtick = ltick - tick;
  ltick = tick;

  /* set MSB to reflect state of input line */
  if ((inportb(comport + 6) & 0xF0) > 0) dtick = dtick | 0x8000;
```

```
                              else dtick = dtick & 0x3fff;

  fdata[cpstn] = dtick;        /* put freq in fdata array              */
  cpstn  ++;                   /* increment data buffer pointer        */
  if (cpstn>buflen) cpstn=0;   /* make sure cpstn doesnt leave array   */

  d1 = inportb (comport + 2);   /* clear IIR                            */
  d1 = inportb (comport + 5);   /* clear LSR                            */
  d1 = inportb (comport + 6);   /* clear MSR                            */
  d1 = inportb (comport);       /* clear RX                             */
  outportb (0x20, 0x20);       /* this is the END OF INTERRUPT SIGNAL */
                               /* "... that's all folks!!!!"          */
}

/*********************************************************************/
/*                   SERIAL PORT INITIALIZATION                      */
/*********************************************************************/
/* basic purpose: enable modem status interrupt and set serial port    */
/*                output lines to supply power to interface            */
void set8250 ()                /* sets up the 8250 UART               */
{
  static unsigned int t;
  outportb (comport+3, 0x00);  /*  set IER on 0x03f9                  */
  outportb (comport+1, 0x08);  /*  enable MODEM STATUS INTERRUPT      */
  outportb (comport+4, 0x0a);  /*  push up RTS, DOWN DTR              */
  t = inportb(comport + 5);    /*  clear LSR                          */
  t = inportb(comport);        /*  clear RX                           */
  t = inportb(comport + 6);    /*  clear MSR                          */
  t = inportb(comport + 2);    /*  clear IID                          */
  t = inportb(comport + 2);    /*  clear IID - again to make sure     */
}

/*********************************************************************/
/*                     TIMER CHIP INITIALIZATION                     */
/*********************************************************************/
/* purpose: make sure computer timekeeper is set up properly. This     */
/*          routine probably isn't necessary - it's just an insurance  */
/*          policy.                                                    */
void set8253()                 /*  set up the 8253 timer chip        */
{                              /* NOTE: ctr zero, the one we are using*/
                               /*  is incremented every 840nSec, is  */
                               /*  main system time keeper for dos   */
  outportb (0x43, 0x34);       /* set ctr 0 to mode 2, binary        */
  outportb (0x40, 0x00);       /* this gives us the max count        */
  outportb (0x40, 0x00);
}

/*********************************************************************/
/*          minor forward error correcting stuff stuff...            */
/*********************************************************************/
/* do error correcting stuff */
```

```
/* matrix for block encoding */
int h3 = 0xEC, h2 = 0xD3, h1 = 0xBA, h0 = 0x75;

/* returns the number of ones in the byte passed to routine */
int ones(int h)
{
  static int i,nb;
  nb = 0;
  for (i=0; i<8; i++)
  {
    if ((h & 0x01) == 1) nb++;
    h = h >> 1;
  }
  return(nb);
}

/* returns number of ones in the integer passed to routine */
int ones_int(int h)
{
  return(ones(h) + ones(h >> 8));
}

/*****************************************************************/
/*          RUN THROUGH BLOCK FORWARD ERROR CORRECTION CODE      */
/*****************************************************************/
/* PURPOSE:                                                      */
/*       does FEC on the 8 data bits and 4 FEC bits passed to it in goi */
/*       blert tries to check and correct errors... If the errors are   */
/*       uncorrectable blert returns a 1 ; otherwise 0                  */
/*       but don't read too much into this - two thirds of the time     */
/*       random crap going into this routine will not generate the      */
/*       uncorrectable error signal... Please Rely on the CRC check     */
int blert(int *goi)
{
   static int dab,nb,bb,uce=0;

   uce = 0;  /* flag that indicates an uncorrectable error */

   /* calculate ecc bits from our current info bits */
   dab = *goi >> 4;
   nb  = (ones(dab & h3) & 0x01) << 3;
   nb += (ones(dab & h2) & 0x01) << 2;
   nb += (ones(dab & h1) & 0x01) << 1;
   nb += (ones(dab & h0) & 0x01);

   /* get syndrome */
   nb = nb ^ (*goi & 0x0f);

   if (ones(nb) > 0) uce = 1; else uce = 0;
   /* if syndrome is not equal to zero try to correct the bad bit */
```

```
   if (ones(nb) > 1)
   {
     if ((nb & 0x08) > 0) bb  = h3; else bb  = h3 ^ 0xff;
     if ((nb & 0x04) > 0) bb &= h2; else bb &= h2 ^ 0xff;
     if ((nb & 0x02) > 0) bb &= h1; else bb &= h1 ^ 0xff;
     if ((nb & 0x01) > 0) bb &= h0; else bb &= h0 ^ 0xff;

     /* are we pointing to a single bit? if so we nailed the bastard */
     if ( ones(bb) == 1) *goi = *goi ^ (bb<<4); else uce ++;
   }
   else *goi = *goi ^ nb;
   /* single wrong bit in syndrome => error occured in FEC bits */

   return(uce);
}

/*****************************************************************/
/*               BIT SCRAMBLING SEQUENCE GENERATOR               */
/*****************************************************************/
/* following is the pseudo-random bit scrambling generator.      */
/* It's the output of a 9 stage linear-feedback shift register with taps */
/* at position 5 and 9 XORed and fed back into the first stage.  */
/* An input of less than zero resets the scrambler.              */
/* Otherwise it returns one bit at a time each time it is called */
int bs(int st)
{
  static int rs,ud;

  /* leave if system isn't supposed to use bit scrambling */
  if (bitscr == 0) return(0);

  if (st < 0) rs = 0x1E;  /* inputs <0 reset scrambler */
  else
  {
    if ( (rs & 0x01) > 0) ud=1; else ud = 0;
    if ( (rs & 0x10) > 0) ud = ud ^ 0x01;
    rs = rs >> 1;
    if (ud > 0) rs = rs ^ 0x100;
  }
  return(ud);
}

/*****************************************************************/
/*                        CRC GENERATING ROUTINE                 */
/*****************************************************************/
/* CRC generator - passing a -1 to this routine resets it, otherwise */
/* pass all 144 data bits in the mobitex data block to it (starting  */
/* with byte 0, LSB bit). The returned value will then be the        */
/* CRC value. Passing any other negative value just returns CRC.     */
unsigned int crc(signed int gin)
{
```

```
  static unsigned int sr = 0x00,cr;

  if (gin >= 0)
  {
    if (gin == 1) cr = cr ^ sr;
    if ((sr & 0x8000) != 0) sr = (sr << 1) ^ 0x0811; else sr = sr << 1;
  }
  else if (gin == -1)                    /* -1 resets the crc state */
  {
    sr = 0xF8E7;
    cr = 0x2A5D;
  }

  return(cr);
}


/*****************************************************************/
/*          PROCESS RECEIVED 240 BIT MOBITEX DATA BLOCK          */
/*****************************************************************/
/* process MOBITEX data block */
int barfrog()
{
  static int i,j,k,b,nerr,cb;
  static unsigned int crcc=0x0000;
  static int blob[30];
  nerr = 0;

  /* process data block into 20 byte chunk stored in array blob */
  crc(-1);            /* reset crc routine */
  crcc = 0x0000;
  for (i=0; i<20; i++)
  {
    /* uninterleave the data into b (holds 8 data bits + 4 FEC bits) */
    b = 0;
    for (j = 0; j<12; j++)
    {
      b = b << 1;
      k = (j*20) + i;
      if (ob[k] == 49) b ^= 0x01;
    }

    /* run through error correction routine */
    nerr+=blert(&b);

    /* spit out data bits to CRC routine - LSB data bit first... */
    cb = b >> 4;
    if (i < 18)
    {
      for (j=0; j<8; j++)
      {
```

```
      if ( (cb & 0x01) == 1) crc(1); else crc(0);
      cb = cb >> 1;
    }
  }
  else
  {
    crcc = (crcc << 8) ^ (cb & 0xff) ;
  }

  /* store the byte in our wonderful wonderful array */
  b = b >> 4;
  blob[i] = b;
}

/* at this point array BLOB holds the data; nerr gives the number  */
/* of errors detected by the FEC code ('corrected' errors count as */
/* one, uncorrectable count as 2); and crcc gives the received     */
/* CRC code. We use this info to decide if we got a good block     */

/* if CRC is correct or nerr <15 we'll say it's a good block */
if ( (crc(-2) == crcc) | (nerr < 15) )
{
  for (i=0; i<20; i++)
  {
    printf("%02X",blob[i]);
    if (outfil) fprintf(out,"%02X",blob[i]);
  }
  if ( crc(-2) == crcc)
  {
    printf ("         ");
    if (outfil) fprintf (out,"         ");
  }
  else
  {
    printf(" BAD CRC ");
    if (outfil) fprintf(out," BAD CRC ");
  }
  for (i=0; i<20; i++) if (blob[i] > 31)
  {
    printf ("%c",blob[i]);
    if (outfil) fprintf (out,"%c",blob[i]);
  }
  else
  {
    printf (".");
    if (outfil) fprintf(out,".");
  }
  printf("\n");
  if (outfil) fprintf(out,"\n");
}
```

```
  return(nerr);
}

/*****************************************************************/
/*              FRAME SYNCHRONIZATION OF RAW BIT STREAM          */
/*****************************************************************/
/* this routine tries to achieve frame sync up in the raw bit stream    */
int frame_sync(char gin)
{
  static int s1=0x0000,s2=0x0000;
  static int cb1,cb2,bc=0,nbc=0,og,nu,bcb,fss = 0,fsb;

  fss = 0;
  /* nbc is a bit counter for # of bits left to form into a data block */
  if (nbc == 0)
  {
    /* nbc = 0 so we aren't trying to process a data block; instead try */
    /* to sync up with incoming bit stream                              */

    /* keep sliding buffers up to date */
    s1 = s1 << 1;
    if ( (s2 & 0x8000) != 0) s1++;
    s2 = s2 << 1;
    if (gin == 49) s2++;

    /* check for sync */
    fsb = ones_int(s1^btsync);
    if (cfs) fsb += ones_int(s2^frsync);
    /* if first two integers match up within a bit or two then  */
    /* we've gotten frame sync                                  */
    if ( (fsb < (1+cfs)) && (bc == 0) )
    {
      if (verbose)
      {
        printf("SYSTEM FRAME SYNC %04X  ;",s2);
        if (outfil) fprintf(out,"SYSTEM FRAME SYNC %04X  ;",s2);
      }
      bc = 25;
    }

    /* bc is a bit counter used to pick off the two status bytes and  */
    /* the FEC byte in the header.                                    */
    if (bc > 0)
    {
      bc--;
      if (bc == 0)
      {
        /* strip off control bytes, run them through FEC routine */
        cb1 = (s1 & 0xff) << 4;
        cb1 += (s2 >> 4) & 0xf;
        cb2 = (s2 >> 4) & 0xff0;
```

```
        cb2 += (s2 & 0xf);
        bcb = blert(&cb1) + blert(&cb2);
        if (bcb == 0) fss = 1;
        if (verbose)
        {
          printf ("Control Bytes #1=%02X, #2=%02X",cb1>>4,cb2>>4);
          if (outfil) fprintf (out,"Control Bytes #1=%02X, #2=%02X",
                                                  cb1>>4,cb2>>4);
          if (bcb != 0)
          {
            printf (" BAD Control Byte FEC");
            if (outfil) fprintf (out," BAD Control Byte FEC");
          }
          printf("\n");
          if (outfil) fprintf(out,"\n");
        }

        /* for RAM network -                                              */
        /*  control byte2 : bits 1 and 2 both 0 -> data block(s) follow */
        /* otherwise just be adventurous and see if we get a valid block */
        if (ramnet) { if ( (cb2 & 0x60) == 0) nbc = 1440; }
                    else nbc = 1440;
        nu = 0;
        bs(-1);
      }
    }
  }
  else
  {
    if (bs(1) == 1) og = gin ^ 0x01; else og = gin;
    ob[nu] = og;
    nu ++;
    if (nu == 240)
    {
      nu = 0 ;
      if (barfrog() < 15) nbc = 241; else nbc = 1;
    }
    nbc --;
    if (nbc == 0)
    {
      printf ("\n");
      if (outfil) fprintf(out,"\n");
    }
  }
  return (fss);
}

/*****************************************************************/
/*                    RECEIVE CLOCK TWEAKING ROUTINE             */
/*****************************************************************/
void check_clk(int i, int nbs)
```

```
{
  static int nt,ii,ndt,na=0;
  static double a=0.0,avg,cvg;

  if (clocka)
  {

    ii = i-1;
    if (ii < 0) ii = buflen;
    nt = 0;
    ndt = 0;
    while (ndt < (54-nbs) )
    {
      nt += fdata[ii];
      ndt = 0.5 + (nt/odtt);
      ii--;
      if (ii < 0) ii = buflen;
    }
    cvg = (double) nt / ndt;

    /* update average if in roughly the right range */
    if ( (cvg < ul) && (cvg > ll))
    {
      a = a + cvg;
      na ++;
      avg = a / (double) na;
      dtt = avg;
    }
  }
}

/*********************************************************************/
/*                          DISPLAY HELP SCREEN                      */
/*********************************************************************/
void help()
{
  printf("\n  MOBITEX RECEIVING PROGRAM - Command line arguement summary\n");
  printf("      (Defaults are set up for RAM Mobile Data's system.)\n\n");
  printf("  /V:1        - print received system frame sync and control blocks\n");
  printf("  /V:0        - don't show the above (DEFAULT)\n");
  printf("  /CFS:1      - check system specific frame sync field (DEFAULT).\n");
  printf("  /CFS:0      - Don't check it - useful for unknown systems.\n");
  printf("  /FS:XXXX    - set system specific frame sync to HEX XXXX (DEFAULT B433)
\n");
  printf("  /SY:YYYY    - set bit sync to HEX YYYY (Default for base stations CCCC)
\n");
  printf("  /BR:Z       - Z is the Mobitex system's baud rate (Default = 8000).\n");
  printf("  /BS:1       - System uses bit scrambling (DEFAULT)\n");
  printf("  /BS:0       - system does not use bit scrambling\n");
  printf("  /CA:1       - program attempts to fine tune receive clock (DEFAULT)\n");
  printf("  /CA:0       - program does not attemp to fine tune receive clock.\n");
```

```
  printf("  /BI:1       - Inverts incoming raw data.\n");
  printf("  /BI:0       - don't invert incoming raw data (DEFAULT).\n");
  printf("  /RN:1       - RAM only: control bytes determine if data blocks follow
sync.\n");
  printf("  /RN:0       - always try to decode data blocks regardless of control
bytes.\n");
  printf("  /OF:1       - Write data to output file (DEFAULT).\n");
  printf("  /OF:0       - Don't write output data file.\n");
  printf("  /COM:z      - set z = 1,2,3,4 to set com port you want to use.\n");
  printf("\nSee text file for further details...");
  printf(" press any key to continue...\n");
  getch();
}

/*********************************************************************/
/*                            MAIN ROUTINE                           */
/*********************************************************************/
void main (int argc,char *argv[],char *env[])
{
  unsigned int n,i=0,j=0,k,l,nbs,imax=0;
  int sport=1,irqv=0x0c;
  char s=48,temp[20],yon[2][5]= {"OFF","ON"};
  double pl,dt,exc=0.0,clk=0.0,xct;

  /* process command line arguements */
  for (n=1; n<argc; n++)
  {
    strcpy(temp,argv[n]);
    strupr(temp);
    j+=sscanf(temp,"/FS:%X",&frsync);
    j+=sscanf(temp,"/SY:%X",&btsync);
    j+=sscanf(temp,"/V:%i",&verbose);
    j+=sscanf(temp,"/CFS:%i",&cfs);
    j+=sscanf(temp,"/BR:%g",&brate);
    j+=sscanf(temp,"/BS:%i",&bitscr);
    j+=sscanf(temp,"/CA:%i",&clocka);
    j+=sscanf(temp,"/BI:%i",&bitinv);
    j+=sscanf(temp,"/RN:%i",&ramnet);
    j+=sscanf(temp,"/OF:%i",&outfil);
    j+=sscanf(temp,"/COM:%i",&sport);
    if (temp[1] == 'H') j=30;
  }

  if ( (j+1) != argc) help();
  printf ("\n\nProgram Options set as follows: \n");
                printf ("Baud Rate        : %g baud\n",brate);
                printf ("Frame sync       : %04X\n",frsync);
                printf ("Bit sync         : %04X\n",btsync);
  cfs    &= 0x01; printf ("Check frame sync : %s\n",yon[cfs]);
  verbose&= 0x01; printf ("Verbose mode     : %s\n",yon[verbose]);
  bitscr &= 0x01; printf ("Bit scrambling   : %s\n",yon[bitscr]);
```

```
clocka &= 0x01; printf ("Fine tune clock  : %s\n",yon[clocka]);
bitinv &= 0x01; printf ("Bit Inversion    : %s\n",yon[bitinv]);
ramnet &= 0x01; printf ("RAM network mode : %s\n",yon[ramnet]);
outfil &= 0x01; printf ("Output file echo : %s\n",yon[outfil]);
sport  &= 0x03; printf ("Running  on  COM%i\n\n",sport);

if (outfil)
{
  strcpy (temp,"MOBYDAXXXXXX");
  mktemp(temp);
  printf("\nfilename where screen display will be echoed : %s\n\n",temp);
  out = fopen(temp,"wt");
}

printf ("Press any key to stop program...\n\n");

/* dtt is the number of expected clock ticks per bit                */
dtt =  1.0/(brate*839.22e-9);      /* this will be the updated clock */
odtt = dtt;                              /* this oddt is not updated */
/* following give the range over which the the clock can be adjusted */
ul = odtt + 1.0;
ll = odtt - 1.0;

/* set up serial port and related parameters                        */
n = inportb (0x21);
if ( (sport == 1) | (sport == 3))
{
  irqv = 0x0c;
  oldfuncc = getvect(irqv);    /* save COM  Vector                  */
  setvect (irqv, com1int);     /* Capture COM  vector               */
  outportb(0x21, n & 0xef);
  if (sport == 1) comport = 0x3f8; else comport = 0x3e8;
}
else
{
  irqv = 0x0b;
  oldfuncc = getvect(irqv);    /* save COM  Vector                  */
  setvect (irqv, com1int);     /* Capture COM  vector               */
  outportb(0x21, n & 0xf7);
  if (sport == 2) comport = 0x2f8; else comport = 0x2e8;
}

set8253();                  /* set up 8253 timer chip              */

set8250();                  /* set up 8250 UART                    */

while ( (cpstn < 3) & (kbhit() == 0) );
if (cpstn < 3)
printf("HEY - no data seems to be coming in over your interface.\n\n");
else
printf("Interface seems to work properly...\n\n");
```

```
  /* main process routine - keeps on going until any key was hit      */
  while (kbhit() == 0)
  {
   if (i != cpstn)
   {
    if  ( ( fdata[i] & 0x8000) != 0) s = 48; else s = 49;
    s = s ^ bitinv;

    /* add in new number of cycles to clock  */
    clk += (fdata[i] & 0x7fff);
    xct = exc + 0.5 * dtt;  /* exc is current boundary */
    nbs = 0;
    while ( clk >= xct )
    {
      /* if frame_sync returns a 1 it means the last 56 bits were the sync */
      /* frame_sync(s); */
      nbs ++;
      if (frame_sync(s) == 1) check_clk(i,nbs);
      clk = clk - dtt;
    }
    /* clk now holds new boundary position. update exc slowly... */
    /* 0.005 sucks; 0.02 better; 0.06 mayber even better; 0.05 seems pretty good */
    exc = exc + 0.050*(clk - exc);

    i++;
    if( i >buflen) i = 0;

    if ( ((cpstn - i) > imax) && (cpstn > i)) imax = cpstn - i;
   }

  }

  /* shutdown */
  outportb (0x21, n);          /* disable IRQ4 interrupt             */
  setvect (irqv, oldfuncc);    /* restore old COM1 Vector            */

  printf (" %i\n",imax);
  if (outfil) fclose(out);
}

/*
```