LAST free COPY

TI-LINES

Volume 1, Issue 2,   MAY 1st., 1984

TI-LINES is the monthly newsletter of the OXON TI USER group and the presentation of any material herein is copyright of each individual author.

TI-LINES is produced and published by:

Peter G. Q. Brooks
29 Kestrel Crescent
Blackbird Leys
OXFORD OX4 5DY

Telephone OXFORD 717985 (after 7 p.m.)

Every effort is made to ensure that the information given in this newsletter is correct. The publisher cannot be held responsible for any inaccuracies.

TI-LINES is also available to blind or partially-sighted Users on audio cassette - contact the publisher for further details.

Contributors to TI-LINES should ensure that their material is submitted either as typed copy or in clear handwriting, where it will be subject to limited editing and retyping.

Submissions should arrive by the first day of the month prior to publication. It is the responsibility of the author to ensure that no copyright infringement will occur by the publication of any material contained in their article.

E D I T O R I A L   C O M M E N T S

I was chatting to a pharmaceutical rep the other day at the hospital where I work, and from him I learned the Gospel According To Mike (Horwood-Smith). "If you invite 4000 people to a meeting, 400 will respond, 40 will say that they are interested, 4 will tell you that they are definitely coming, and one will turn up."

Out of the 25 people to whom I originally sent the circular concerning the User group, 8 responded over nine weeks expressing a strong interest, and after the first issue, one sent me his stamps (and preferred to pay the total cost of the newsletter; however, while this is fine for those working, those at school and those unemployed might find it difficult to cover the cost - see later). The first copy cost 65p to produce, excluding materials and postage (i.e., photocopying cost was 65p), and I can see that future copies will become increasingly expensive. However, based on Mike's Gospel I don't expect that all 25 will take up the subsidised subscription, so I am quite happy to continue covering all but the postage cost for the foreseeable future. If the Elite Eight take out a subscription it will cost me the vast sum of about £6 a month - an amount which I (theoretically) earn from Home Computing Weekly for a week's software reviewing - and for the opportunity to shoot my mouth off in public once again, that is a small price to pay!

I've had one suggestion concerning the format of TI-LINES from fully-paid-up member Jan-Paul Nijman which agrees with my own idea, and that is to eventually produce the newsletter as reduction-Xeroxed A5 in a similar style to that used by Tidings (gone but not forgotten) and TI.MES. I've a feeling though that the cost could soar considerably unless I can put in an order for a sizeable number of copies. That's something to look at in the near future.

Something else to look at will be the possibility of carrying advertising, which could eventually cover the cost of publication and thus place TI-LINES in the same league as the 'free' newspaper. I have been compiling a list of soft- and hard-ware suppliers who might be approached once the group has achieved some stability, and I have decided to allow owners outside Oxfordshire to subscribe if they so wish. A number of the local American User groups mail their newsletters all over the world, and the material in them is largely concerned with local events, with little of the reference material which I hope to fill TI-LINES with over the coming months.

Some of you might have wondered where the title TI-LINES comes from; I came across it while reading a Texas manual on the 990 minicomputer series. It is an Input/Output facility on the 990/5 and 990/10 only, and the connection seemed too good not to make use of it. Any riveting alternatives will naturally be considered.

This second issue sees a continuation of the article on the Control and Function keys with a discussion of tokens amongst other things; an article from outside the group: Dave Hewitt of the Hoddesdon User group on making use of your own TV or video UHF modulator; and the beginning of the BEGINNER'S BASIC which first appeared in Tidings. This series made reference to another which appeared in Tidings: the Maths revision series which may, if necessary, be brought in to swell the ranks as it were. We will also be examining some of the data being produced by Richard Blanden, who appears to have delved deeper into the 4A than anyone before him. Later, if time permits, we will have an article concerning problems experienced with a continental TV by GRAHAM DIACON.

Finally, there is a short questionnaire enclosed (with postage materials) which, if you are feeling helpful, you might fill in. All replies will be treated in the strictest confidence - in fact, once I have the necessary numerical data, I will be destroying the returned sheets. I look forward to hearing from you.

2021- nb: vital information is omitted- see the ERRATA on p9 of the next issue.
This information applies to modulators in a METAL case not the plastic housed modulators. bb

M O D I F Y   Y O U R   M O D U L A T O R

by DAVE HEWITT (originally published under BABBLING BROOKS II in TI.MES, Spring 84)

For some time we have been wondering about the possibility of connecting a TI-99/4A directly to our home video recorder, or for that matter directly to a TV monitor (i.e. RGB + composite video).

You may have noticed that many new domestic TV sets now come equipped with sockets at the back to accept direct sound and video inputs, and in other models such as the THORN TX10 chassis range (not the TX9 or 90 range however) a kit to adapt them is available from the manufacturer's Spares department.

We looked at the video modulator circuits for other home computers and at a MULLARD circuit for converting the type of signal produced by the 99/4A (i.e. R-y, B-y, + composite video. We understand that this somewhat odd arrangement for outputting video from a computer is a hang-over from the machine's beginnings as an NTSC model, or so an item in TIHOME's Tidings suggested.)

While contemplating the problem of making a unit which would encode this signal, we wondered how Texas had overcome the difficulty. If you think about it (we would be embarrassed to tell you how long it was before the penny dropped!) something of the sort has to go on inside the UHF modulator (that little box which goes between computer console and TV).

In fact, several stages of conversion and mixing (matrixing) go on inside this unit. The colour information is added to the black and white (or, if you like, the composite video has the chrominance information encoded on it) plus the sync-pulses (to tell the TV where to start scanning and when to stop, etc.). There is also one other vital signal which is added right at the end, and that is on a separate sub-carrier of 6 Mcycles, and this is the audio or sound signal.

When all this information has been processed and received some amplification it is finally passed on to the UHF modulator itself. An Ultra High Frequency modulator is the part of the circuit which takes the mentioned signal package and converts it into the sort of output that your UHF TV aerial input socket is looking for. (We could go on about modulation for a lot longer, but it is already getting out of hand!).

To cut a long story short, what you need to do is to disconnect the modulator from the console and the TV and remove the three screws which hold the top cover on. The part of the circuitry which we want to get at is the input into the UHF modulator from the processing circuits which preceed it. This signal is at just the right level and impedance to feed into the already-mentioned video input circuit on a TV set or into the camera or auxiliary socket which is provided on most video recorders. We have found that the audio circuit can be directly coupled to the audio input, but the video circuit needs coupling through a 47 uF condenser (in practice any condenser we tried between 10 uF and 1000 uF worked fine). With the modulator unit turned so that the lead to the computer is coming out of the bottom, observe this lead as it enters the box, and you will find that its inner cables divide 6 ways (including the outer screen which has been made tidy by wrapping in a bit of thick yellow sleeving). The audio signal is carried by the thin yellow wire which goes to the bottom right hand side of the printed circuit board.

Cut this wire and fit a suitable piece of sleeving over it (to insulate the connection again when you have finished). Join a fresh length of suitable audio single screened wire to the junction of these wires, the inner cable being joined now to the yellow audio lead. Fold back the outer screened wire of the new cable, fit some sleeving over this and solder the end on to a convenient point on the metal case of the box (the silver-coloured screening can of the modulator is ideal for this purpose).

Fit the correct type of plug on the other end of the new cable to fit the audio input of your TV or video. This lead may be used on its own for taking the output of the computer's sound circuits to your stereo system, to give a better response that the speaker on your TV might offer.

If you wanted to be really clever you could add a volume control at this point, but we will not go into that now.

Next locate the small 1 kilohm resistor which sits above and to the right of the modulator. Solder another inner cable from the left hand side of this resistor, again soldering the outer screening cable onto the modulator's screening can. Fix a suitable plug on the other end of this lead and fit into the video input socket of your video or TV.

Switch your video to the camera or aux. position and you can observe the output from your computer via the channel which you would normally use for playing video recordings.

This now allows you to make video recordings of your favourite video games, and to have 'action replays'. Just think of that high score in your Alien-Mothers-in-Law program which nobody would believe - well, now you can prove it!

In general we have found that the quality from the more sophisticated modulator in your video is higher than that obtained from the unit supplied with the micro. We use this direct-input facility to record professional titles on our home video recordings, to do simple graphic headings, and even record some of the visually-pleasing patterns produced by the Designs program given in an edition of Tidings by Pete Brooks.

If you connect the inputs directly into the sockets on the back of a domestic TV the gain in picture definition and stability is quite marked, and frees you from that annoying tuner drift which often occurs as modulators and tuners warm up.

All that remains is to drill two holes in the modulator unit's casing, large enough to accept the two new cables with protecting grommets. If preferred, one could fit suitable chassis sockets on the lid or case of the TI modulator so that the unit can be unplugged from the new leads, or maybe a line socket could be used on a short lead, but both of course should use proper co-axial screened cable.

We have attached 12 feet of lead on both outputs and have not encountered any undue 'pick-up'. The UHF output from the TI modulator is not disturbed and may be used at the same time without affecting either cable.

Editor's note: if you decide to follow the instructions given in this article, please take every precaution possible to guard against any chance of electric shock. Any reader undertaking the modification discussed here does so at his own risk. The publisher and the author cannot be held responsible for any damage which may ensue as a result of this modification.

- -

Both the editor of TI.MES, Clive Scally, and I have tried to persuade Dave Hewitt to write regularly on hardware modifications which can be undertaken by TI-99/4A owners in order to improve their systems, but in vain. Dave cannot find the time to make a regular contribution. However, as a result of a recent telephone chat to Dave I may be able to present another article in the near future which will enable all those possessing a disk system to access BOTH sides of a disk. Note that this will require DOUBLE-SIDED DISKS, as the standard system uses only SINGLE-SIDED. Dave assures me that the modification is ridiculously simple, involving the use of a single resistor to permit DSK1. and DSK2. accesses.

B E G I N N E R ' S   B A S I C   I I

by Peter Brooks (originally published in Tidings, V2.4, August 82)

I n t r o d u c t i o n

Although TI manuals have a reputation for being comprehensive and well-written, it would appear that many owners have been experiencing some difficulty in following the explanations given. It may be that this difficulty arises because the manuals assume some basic understanding on the part of the reader, or, more likely in my view, that the majority of readers are not used to reading what amount to reference works. There is a degree of skill involved in sifting through and absorbing the small amount of information required from the mass presented; this problem was highlighted in a questionnaire sent out by TIHOME in 1982, in which many responses indicated that BEGINNER'S BASIC would need to be presented in a very elementary format, because many readers were putting the "Babbling Brooks" to one side against the day that they felt equipped to understand them. (Babbling Brooks was the title of a regular series which was comprised of several different items, all of them aimed at the novice user.)

In the light of that feedback I started the Beginner's BASIC, aimed at the novice novice (yep, two novices) user. The intention was not to replace the TI manuals, but to supplement them, as the same information, when presented several different ways, stands a better chance of being fully understood.

And if you think that my explanations don't improve matters, don't forget to write in and air your views.

Back in the early days of computing, you really did need to be something of a brain-box in order to program a computer. The early machines were programmed in binary code, by setting banks of switches; a far cry from today's sophisticated machines, some of which allow you to program verbally, by accepting speech input.

Someone had a brainwave, and made life a little easier, by allowing programmers to enter their programs using either hexadecimal (base 16) or octal (base 8) code, through simple keyboards. Later, the so-called 'high level' languages were developed, and still are being developed. These allowed programmers to enter what amount to English words, and one of the early languages was ForTran - Formula Translation,a 'scientific' language, from which BASIC (Beginners All-purpose Symbolic Instruction Code) was developed in about 1957 at Dartmouth College, USA.

BASIC was apparently intended originally to help engineering and computer science undergraduates understand the principles of programming; a kind of 'Janet & John' language which could be used to build up to the more complex languages used.

But, BASIC proved to be so popular, precisely because it is so easy to learn and use, that it is still the predominant language around today, despite the fact that much more powerful languages have since been developed. In other words, it is as if most people are still using 'goo-goo ga-ga' baby language as their main means of communication, which is a source of irritation to all academics and some of the 'professionals'. This is why you will see articles in the popular computing press which decry the amateur programmer and his BASIC, and why you won't stand much chance of getting a job in the computer industry if you have only BASIC under your belt.

To the average amateur, however, BASIC doesn't seem much like a simple language, until they see programs written in Assembly Language or Pascal, or even Forth.

One hurdle that I found was that I couldn't grasp how the computer actually understood BASIC.

Most books will tell you that computers only 'understand' machine code (which can mean binary code, octal, hexadecimal, or even something called microcode!). Even now, I have gained only a broad understanding of what goes on, and I have found that the key to understanding BASIC is a little understanding of what goes on 'behind the scenes' - in other words, how the computer 'sees' things.

To begin with, from the moment that we switch the computer on, programs are running. There is a program which prepares and produces the screen display, sending it to the TV together with instructions for synchronisation. There is a program which checks the keyboard to see which keys are being pressed and what should be done if any are. There are others which actually execute the BASIC programs which you write, and yet others which handle the flow of information between computer and peripherals.

Unfortunately, even today, very few details are available about the internal workings of the 99s, so one can only speculate in some cases about what goes on inside. You can get an inkling of the volume of work being done by the machine if, after selecting TI BASIC, you turn up the volume of the TV (hopefully properly tuned to the right channel) when you will be able to actually 'hear' the cursor blinking on and off. If you then press a key like ENTER for example, you will hear the tone change as the computer responds. Type a valid instruction like CALL CLEAR and enter it (and make sure it IS correct or you'll blow your socks off with a deafening error tone!) and you can hear the machine at work. When you enter something, by pressing ENTER or using one of the other entry keys: FCTN X or FCTN E (replace FCTN with Shift on 99/4s), the computer runs part of a program which compares what you have typed with what amounts to a list or dictionary of instructions which it can 'understand'; if it finds that what you have entered doesn't appear in the list, it runs another part of a program which deals with errors. Otherwise it will execute whatever machine code routines (and there are many) it needs to use in order to carry out the commands which you have issued.

When you enter a BASIC program, there is a lot which you the User do not realise is happening. When you type a line number followed by a BASIC statement, and then ENTER it, the computer runs a general check to see if what you have entered is allowed by the rules which have been laid down by the designer. Try entering

```
        100  RUN
```

and see if it is accepted. TI BASIC won't allow RUN to be used as a statement in a program. Assuming that the line you have typed is not replacing one which exists already, the computer then replaces all the RESERVED WORDS (like PRINT, IF, FOR, INPUT, etc.) with TOKENS, which are single characters and therefore take up far less room in memory. Certain other symbols (like =, +, &, -, etc.) are also replaced. This aspect of the operation of the machine is being covered in issues 1 and 2 of TI-LINES.

There are at least two facets to any attempt to write a program. One is the language used - in this case BASIC, which will be looked at in greater detail later. The other is probably the most difficult thing to understand or to explain, and that is what it actually means to 'program'. I am handicapped by the fact that it is now 7 years since I first "learned" to program, and I find it very difficult to remember exactly what it was like before then.

I have yet to find a simple way of getting the novice programmer to realise that he or she has been programming for years without ever being conscious of it. If you have ever written a shopping list you've written part of a program; every waking hour is filled with little programs which you have learned over the years: ones which make you get dressed before going out, which make you open your mouth before shovelling food in, which make you open a door before trying to go through it. In theory, anyway. This doing things the right way round is an example of something called an ALGORITHM, a rule or set of rules which are used to achieve an objective, and is an integral part of any program.

I have tried giving programming examples which make people think: my favourites are of the smoking and shopping variety. You either write down the sequence of actions involved in lighting up, or in sending the kids to the shops for a tin of beans. The trouble is that while these illustrate the principles of programming and of logical thinking, they have no relevance to BASIC, unless you happen to have a robot tacked onto your 99. The above examples are concerned more with the control of things, whereas BASIC is more concerned with processing data.

A better example might be if I was to ask you to do some arithmetic for me. If I asked you to add four numbers together, how would you set about it ? You might ask for the first number, and then perhaps write it down. You would then ask for the second number, and write that down under the first, and so on. Once you had the four numbers, you'd add them up, writing the answer underneath. You might then tell me what it was.

There are three separate actions occurring here. The first is INPUT, the second is PROCESSING DATA, and the third is OUTPUT. Things are not always so cut and dried, as you will come to find out; sometimes processing halts for further input, sometimes there is no output, sometimes no input as such. But generally programs can be divided up into those three sections. The sequence has to be right, too, before the program will work properly. You couldn't give me the answer BEFORE I'd given you the numbers, for example. The INPUT section is fairly obvious: it consists of you prompting me for the numbers, and me giving them to you when you are ready for them. The PROCESSING is also fairly obvious - it consists of you doing the adding up. The OUTPUT consists of you telling me what the answer is.

Not all processing involves arithmetic, however; at least, not in the sense implied above. Suppose that I had asked you to sort some words into alphabetical order instead. Again there would be an INPUT section, where you prompted me for the words, perhaps writing them down on individual cards. The PROCESSING this time would involve shuffling the cards around according to a set of rules (an ALGORITHM, remember ?) until they were sorted. The OUTPUT might consist of you reading the cards out to me, or storing them away somewhere without reading them out at all.

In theory what you are supposed to do is to sit down and specify the problem that you want to solve, or the game that you want to play. You then write down as many of the steps involved as you can, elaborating where necessary. You go over this many times, trying to iron out errors, making sure that you have covered everything (which of course you never have), until you are satisfied that you have done all that you need to. You can then begin to write the program in the language which you have chosen.

What usually happens though is that you sit down at the console with a rough idea of what you want to do, and you begin writing, testing each little bit as you add it, until you have a program which works. Until you present it with something which it can't handle, in which case you now have to go right back to the beginning in order to find out what went wrong, and by now you've forgotten exactly why you did certain things, and you still keep getting INCORRECT STATEMENT IN 460, although 460 looks OK to you... At this point you wish you'd kept notes on what you have been doing, and the three scribbled comments on the back of an old beermat don't mean a thing a fortnight later, and you might as well throw the whole thing over and take up fishing. Nice quiet hobby, fishing.

If you've been bitten by the programming bug though, you'll be back, having been struck by a bolt of wisdom on platform 7 at Reading General, breezing into the house, commandeering the telly in the middle of Coronation Street, and four hours later you're looking out the flies and the keep net...

In time, if you are lucky, you will come to develop a programming style which suits you, and if you are really lucky, one which permits you to program with the minimum of hair-tearing. Don't expect this stage to be reached for quite some time, though, unless you are undergoing tuition. Or reading this series.....

T H E   C O N T R O L   A N D   F U N C T I O N   K E Y S

P A R T   I I

P e t e r   B r o o k s

In the first part of this article we looked mainly at ASCII rather than at using the CTRL and FCTN keys, primarily because before you can discuss the keyboard scans and 'active' keys, you need some basic knowledge as background. So far we have glanced briefly at the function of CTRL and FCTN in their roles as control and editing keys, and at their use to obtain TOKENS. The Users Reference Guide (URG) gives a list of almost all the control characters, but you may have been confused by the references to 'Pascal' and 'BASIC' modes, and the two different sets of key codes listed on page 93. Quite why there should be a difference between the two modes is not clear.

A 'mode', in case you were wondering, refers essentially to the way in which the computer has been programmed to respond according to what it happens to be doing. When you are running a program, the computer is in RUN mode and will not respond in quite the same way as it would in IMMEDIATE mode. Immediate mode is the one you get when you select TI BASIC (or Extended BASIC, for example) - it means that the computer will respond 'immediately' to certain commands, rather than storing them away for later execution (i.e., what happens when you enter program lines).

For example, in RUN mode, pressing and holding down FCTN and then pressing 4 will BREAK the program - the computer will then return to the IMMEDIATE mode. However, if you perform the same key presses in Immediate mode, the computer will 'ignore' any instructions which you have just typed (but not ENTERed), and present you with a fresh line ready for another command.

Try this:- get into Immediate mode, making sure that there is no program currently resident - if necessary type NEW and press ENTER - then press and hold down the CTRL key and press these other keys one after another:

> the comma (,), the letters A to Z, the full stop (.), the semicolon (;), the equals (=), and the digits 8 and then 9.

Release the CTRL key and press and hold down the FCTN key and then press 4 to perform the BREAK. The display will scroll up, a fresh 'prompt' will appear (the 'greater than' symbol), and the cursor will indicate that the computer is ready to receive instructions. If you had pressed ENTER instead of BREAK, the response would have been an error tone and the message BAD NAME - i.e., the computer would NOT have ignored what you had typed.

BASIC mode therefore is what you are in when using TI BASIC, and Pascal mode is what you are in when using the UCSD Pascal system which is available for the TI.

Now thoroughly confused, you probably wonder what on Earth all those control **characters** which you have just entered are supposed to do. Last issue I gave two short, very similar routines to print out on screen all of the User-definable graphics characters (UDGs), and I mentioned that it was possible to use the CTRL key to replace one of the routines, which is what we have now done. You might choose to simplify things and just use CTRL with the alphabet keys (A - Z); the intention is simply to provide a visual indication that any incoming program is being loaded successfully. There is a set of tables later which gives, amongst other things, a list of the ASCII characters 127 to 255 and, where possible, the keys to be pressed to obtain those characters on the screen (in a program listing, for example). In some cases, more than one combination of keys can be pressed - for example to get the character whose code is 133 - and in some instances you may find that your keyboard gives slightly different results: RICHARD BLANDEN tells me that on his 4A FCTN Q gives ASCII 185 instead of 197, which means that his keyboard is probably decoded differently.

The lists contain more information - for example, the TOKENS, which we will examine a little more closely later - but our interest at the moment centres around the UDGs. Having placed them on the screen with the CTRL key, get a program which you have already recorded on tape, and begin the OLDing sequence (see page 9, last issue), just to see the effect.

Once your program has OLDed successfully, type NEW and press ENTER. The UDGs won't now be on screen: place them there as described earlier with the CTRL key. Note that the 'shapes' are still defined. Most, if not all of them, will be blocks of apparently randomly-scattered dots and lines, but what you are actually looking at is part of your BASIC program. To digress a little (again), when you typed NEW and entered it, the computer DIDN'T remove your BASIC program from memory. What it did do was to alter a 'system variable'. System variables are values referred to, and in use by, the computer as it not only runs your programs but also when in Immediate mode, etc. Somewhere in memory the computer stores details about any resident BASIC program: where it is currently stored, for example. One system variable holds the address in memory where the listing begins, another where it ends. If those two system variables hold the same value (so that the 'listing' begins and ends at the same place) then there is no BASIC program on board as far as the computer is concerned. NEW alters the 'end of listing' variable, and if we had been given access to machine code as standard (PEEK, POKE and USR or similar) we would have been able to 'recover' a program if we had inadvertently NEWed it.

Now, because the UDGs definition area in memory, and the TI BASIC program listing area, begin at roughly the same point,(see last issue) we can 'see', to a limited extent, what a BASIC program 'looks like', by examining the UDGs <u>without</u> redefining them. (Again, had CALL CHARPAT() been included in TI BASIC, we could have obtained the current definition strings for the 'undefined' UDGs, and thus examined part of the BASIC program <u>without</u> using machine code commands. You can try this with MiniMemory, but I doubt if it is possible with either Editor/Assembler or Extended BASIC + 32K RAM Expansion, because they don't make use of VDP RAM in quite the same way.)

For example, take the BASIC instruction OPTION BASE - disregard the 0 or 1 for the moment. When stored in the computer's memory, OPTION and BASE are 'tokenised' - that is, ASCII 158 is placed in memory instead of the full word OPTION, and ASCII 241 in place of BASE. In two consecutive memory locations therefore are the binary equivalents of 158 and 241 decimal. In hexadecimal, 158 is 9E, and 241 is F1. So what ? you might say.

Well, try defining a graphics shape to be "9EF1" - you can do it very simply with say CALL CHAR(159, "9EF1") in the Immediate mode, and then use CTRL 9 to place the character whose code is 159 on the screen. The shape you see is OPTION BASE as it appears in memory through our 'window'. Try translating other tokens in the same way, and then go one step further: take a simple phrase like "HELLO MA", work out the ASCII codes for each of the letters and the space, translate them into hex (the text is 8 characters long, and the resulting hex string will be 16 digits long) and then use that with CALL CHAR() to redefine ASCII 159. You should have arrived at "48454C4C4F204041". Put the character on the screen with CTRL 9 to see what 'HELLO MA' will look like when stored internally.

Now, quit, select TI BASIC again, put in this routine, and run it:

```
100 CALL CLEAR
110 DIM T$(128)
120 FOR I=128 TO 159
130 PRINT CHR$(I);
140 NEXT I
150 FOR I=1 TO 128
160 T$(I)="HELLO MA"
170 NEXT I
180 GOTO 150
```

TOKENS IN TI BASIC : KEYBOARD - DIRECT : AFTER STEPHEN SHAW TIDINGS V2.3 1982

ASC	STROKE	FUNCTION	ALTERNATIVE	ASC	STROKE	FUNCTION	ALTERNATIVE
127	FCTN V	Unused (DEL)		164		Unused	
128	CTRL ,	Unused		165		Unused	
129	CTRL A	ELSE		166		Unused	
130	CTRL B	Unused		167		Unused	
131	CTRL C	Unused		168		Unused	
132	CTRL D	IF		169		Unused	
133	CTRL E	GO	CTRL Shift D	170		Unused	
134	CTRL F	GOTO		171		Unused	
135	CTRL G	GOSUB		172		Unused	
136	CTRL H	RETURN		173		Unused	
137	CTRL I	DEF		174		Unused	
138	CTRL J	DIM		175		Unused	
139	CTRL K	END		176	CTRL 0	THEN	
140	CTRL L	FOR		177	CTRL 1	TO	
141	CTRL M	LET		178	CTRL 2	STEP	
142	CTRL N	BREAK		179	CTRL 3	,	
143	CTRL O	UNBREAK		180	CTRL 4	;	
144	CTRL P	TRACE		181	CTRL 5	:	
145	CTRL Q	UNTRACE	CTRL Shift A	182	CTRL 6	)	
146	CTRL R	INPUT	CTRL Shift F	183	CTRL 7	(	
147	CTRL S	DATA		184	FCTN ,	&	
148	CTRL T	RESTORE	CTRL Shift G	185	FCTN .	Unused	
149	CTRL U	RANDOMIZE		186	FCTN /	Unused	
150	CTRL V	NEXT		187	CTRL /	Unused	
151	CTRL W	READ	CTRL Shift S	188	FCTN 0	Unused	FCTN)
152	CTRL X	STOP		189	FCTN ;	Unused	
153	CTRL Y	DELETE		190	FCTN B	=	
154	CTRL Z	REM		191	FCTN H	<	
155	CTRL .	ON		192	FCTN J	>	
156	CTRL ;	PRINT		193	FCTN K	+	
157	CTRL =	CALL		194	FCTN L	-	
158	CTRL 8	OPTION		195	FCTN M	*	
159	CTRL 9	OPEN		196	FCTN N	/	
160		CLOSE		197	FCTN Q	^	
161		SUB		198	FCTN Y	Unused	
162		DISPLAY		199		Quoted string	
163		Unused		200		Unquoted string	

ASC	STROKE	FUNCTION	ALTERNATIVE
201		Line number	
202		EOF	
203		ABS	
204		ATN	
205		COS	
206		EXP	
207		INT	
208		LOG	
209		SGN	
210		SIN	
211		SQR	
212		TAN	
213		LEN	
214		CHR$	
215		RND	
216		SEG$	
217		POS	
218		VAL	
219		STR$	
220		ASC	
221		Unused	
222		REC	
223		Unused	
224		Unused	
225		Unused	
226		Unused	
227		Unused	
228		Unused	
229		Unused	
230		Unused	
231		Unused	
232		Unused	
233		Unused	
234		Unused	
235		Unused	
236		Unused	
237		Unused	
238		Unused	
239		Unused	
240		Unused	
241		BASE	
242		Unused	
243		VARIABLE	
244		RELATIVE	
245		INTERNAL	
246		SEQUENTIAL	
247		OUTPUT	
248		UPDATE	
249		APPEND	
250		FIXED	
251		PERMANENT	
252		TAB	
253		#	
254		Unused	
255		Unused	

- -

Using the CTRL (Control) key, all the User-definable characters may be brought directly onto the screen (and into listings) through the keyboard, rather than by using CHR$() as is necessary on the TI-99/4. During cassette OLDing in TI BASIC, incoming program data is transferred into the same area of memory used to contain definitions of any User-defined characters which may subsequently be used. This explains the appearance of 'pre-defined' shapes reported by some users when pressing either CTRL or FCTN and other keys.

- -

TOKENS : BY RESERVED WORD / SYMBOL

TOKEN	CODE	KEYSTROKE(S)	TOKEN	CODE	KEYSTROKE(S)
ABS	203		OPEN	159	$^c9$
APPEND	249		OPTION	158	$^c8$
ASC	220		OUTPUT	247	($^f/$)
ATN	204		PERMANENT	251	($^c/$)
BASE	241		POS	217	
BREAK	142	^{c}N	PRINT	156	$^c;$
CALL	157	$^c=$	RANDOMIZE	149	^{c}U
CHR∅	214		READ	151	^{c}W , ^{c}shift S
CLOSE	160		REC	222	
COS	205		RELATIVE	244	
DATA	147	^{c}S	REM	154	^{c}Z
DEF	137	^{c}I	RESTORE	148	^{c}T , ^{c}shift G
DELETE	153	^{c}Y	RETURN	136	^{c}H
DIM	138	^{c}J	RND	215	
DISPLAY	162		SEG∅	216	
ELSE	129	^{c}A	SEQUENTIAL	246	
END	139	^{c}K	SGN	209	
EOF	202		SIN	210	($^f/$)
EXP	206	($^f/$)	SQR	211	
FIXED	250		STEP	178	$^c2$
FOR	140	^{c}L	STOP	152	^{c}X
GO	133	^{c}E , ^{c}shift D	STR∅	219	
GOSUB	135	^{c}G	SUB	161	
GOTO	134	^{c}F	TAB	252	
IF	132	^{c}D	TAN	212	
INPUT	146	^{c}R , ^{c}shift F	THEN	176	$^c0$ (zero)
INT	207		TO	177	$^c1$
INTERNAL	245	($^c/$)	TRACE	144	^{c}P
LEN	213		UNBREAK	143	^{c}O
LET	141	^{c}M	UNTRACE	145	^{c}Q , ^{c}shift A
LOG	208		UPDATE	248	
NEXT	150	^{c}V	VAL	218	(^{c}C)
ON	155	$^c.$	VARIABLE	243	

CTRL = ^{c} FCTN = ^{f} Indirect = () string symbol = ∅

TOKEN	CODE	KEYSTROKE(S)	TOKEN	CODE	KEYSTROKE(S)
QUOTED STRING	199	(^{c}C)	&	184	$^{f},$
UNQUOTED STRING	200	(^{f}Y)	=	190	^{f}B
LINE NUMBER	201	(^{c}B)	<	191	^{f}H
#	253	$(^{c}/)$	>	192	^{f}J
,	179	$^{c}3$	+	193	^{f}K
;	180	$^{c}4$	-	194	^{f}L
:	181	$^{c}5$	*	195	^{f}M
)	182	$^{c}6$	/	196	^{f}N
(	183	$^{c}7$	∧	197	^{f}Q

- -

CTRL = ^{c} FCTN = ^{f} Indirect = () string symbol = $

- -

CONTROL/FUNCTION KEYCODES : "UNUSED" RESULTANT SEQUENCES (TI BASIC) : PROGRAM

```
NEW

100  REM A$ = "insert single ctrl/fctn character here"
110  FOR L = 1 TO LEN(A$)
120  PRINT ASC(SEG$(A$, L, 1));
130  NEXT L

EDIT 100

100  delete 'REM' to give A$ = "sequence"

RUN
```

Printout of ASCII codes of sequence occurs

In some instances, 'unused' control/function characters have a use in Extended BASIC, and there may well be additional language modules which employ further 'unused' codes.

- -

The sequence runs like this:

1) Clear all the UDGs of shapes; come back into TI BASIC
2) Clear the screen
3) Reserve space for a string array which has 129 elements (0 - 128)
4) Print out all the UDGs in a continuous string
5) Put the text "HELLO MA" into every element of the array
6) Go back and do (5) again until BREAK is pressed

If you leave this running for some time, you will notice one or two peculiar things happening; see if you can explain what they are and why they occur (answers next time).

Above all else, you should be able to recognise your 'HELLO MA' pattern appearing over and over again in the UDGs, although it probably won't sit squarely in each character - it may overlap from one character to the next. You'll have to BREAK the program to stop it running. Try different groups of letters to see the shapes that different words or sentences produce.

OK, you say, now what ? I still don't see how to use CTRL or FCTN in my own programs.

Well, if you examine the CALL KEY() command in the URG, you'll see that with some keyboard scans the FCTN and CTRL keys are 'active' - they will return a number if you press a combination of either CTRL or FCTN and certain other keys - and in others they are not. If you want to provide some special keys to be pressed, to provide some options for the user, you could use "PRESS 1 FOR this, 2 FOR that," etc., OR, you could use "PRESS CTRL AND 1 FOR this, AND 2 FOR that," etc.

Because the TI use of CTRL characters is non-standard, they will not have the same power as on other systems, but they can be used to extend your range of keys for use as menu options.

However, it must be said that it is probably simpler and easier NOT to use CTRL or FCTN in this way - in menus, you are unlikely to need to use more than the keys A - Z and 0 - 9 for any program that you could write for the 99s.

Where CTRL does have a use is in listings. Instead of using the laborious (and space-consuming) CHR$() function (with or without loops) to manipulate the UDGs, you could employ the characters directly: for example, instead of:-

```
PRINT CHR$(128);CHR$(129);CHR$(130) : CHR$(131);CHR$(132);CHR$(133) :
CHR$(134);CHR$(135);CHR$(136) : : :
```

You COULD shorten it to:-

```
FOR I = 128 TO 134 STEP 3
PRINT CHR$(I);CHR$(I+1);CHR$(I+2)
NEXT I
PRINT : :
```

But better to use:-

```
PRINT  "CTRL , CTRL A CTRL B":"CTRL C CTRL D CTRL E":"CTRL F CTRL G CTRL H": :
```

where, obviously, you don't type C-T-R-L in full, you press and hold down the CTRL key and then press the key for the letter or punctuation mark shown. Don't forget to make a note somewhere of the keys that you have used, and in what sequence, for reference (although you can edit any such line and replace "PRINT" with "A$=", and then encompass the following PRINT statement with quotes. You can then run a loop of 1 to LEN(A$), and PRINT out the ASCII codes - ASC(SEG$(A$, loop, 1)); which will show you which characters are 'hidden' in the listing. Next issue we'll use this technique to uncover some interesting things.).

Note that after the first RUNning of any program containing the UDGs directly in listings, they will 'appear' in their redefined form in the listing. Remember this when producing hard copy on any printer capable of reproducing the listing exactly.

We now come to a more detailed examination of the tables of tokens. I have arranged them by ASCII code and by alphabetical key word. It might be useful here to stop and point out the difference between key words and Reserved words. A key word seems to be,universally, the BASIC word which is replaced by a token when listings are entered ('tokenisation'). A Reserved word is one which has been reserved for the computer's use and the user cannot employ them as variable names. In this respect, LIST, RUN, CON(TINUE) etc., are Reserved words but not tokens, while LET, IF, PRINT, etc., are both key words AND Reserved words. (Just as you cannot have 100 RUN in TI BASIC, you also cannot have 150 CON = 3000 or 1000 FOR IF = THEN TO ELSE STEP GO....1100 NEXT IF!). The URG lists the Reserved words, and in the tables here the key words are given. Notice that there are even tokens for +, -, /, &, (,), etc., and that some key codes are 'unused'. Under certain circumstances, if placed in a listing these will produce gobbledegook in TI BASIC; some DO have a use in Extended BASIC (see Stephen Shaw's list in Tidings) and others have some function but at the moment we know the function and not the language!

Some of them are UPRC$, DAT$, etc., (related to me by Richard Blanden) and I have seen these before in a TI 990 manual. This tends to support the belief that the 99/4 and /4A are cut-down versions of full-blown mini-computers, complete with mini-computer operating systems. (For example, in the FILE PROCESSING commands, the file description PERMANENT is given, suggesting that there is possibly also a TEMPORARY description. The URG says that PERMANENT may be omitted as all 99 files can be considered permanent - if that is the case, why have PERMANENT at all ? Because the mini-computer operating system from which the 4 and 4A system is derived uses PERMANENT and TEMPORARY ?...)

The use of CTRL and FCTN to obtain these tokens is of little real practical help when programming, but it is a tool with which to dig a little deeper into the 99/4A (but not unfortunately the 99/4, which has no CTRL or FCTN keys), without needing to expand it.

Although we will have articles later for those possessing MiniMemory and the like, this initial foray is for those who have just the console and who want to play detective (my favourite past-time bar none!).

In the table of tokens by ASCII code, you'll see that after ASCII 198 there are tokens but no key-strokes. There is also a block between ASCII 160 and 175 which also cannot be accessed directly from the keyboard. (If YOU find anything different, please let us all know.).

In the next issue of TI-LINES we'll look at ways of accessing these 'indirectly' - some initial details are given in the tables - and hopefully play around with the effects of editing lines, and discover the devastating effect of ASCII code 0.

Now for some fun. Clear any program from your computer, type:

```
    1REM
```

(note: no spaces) and then press and hold down the CTRL key, and press U and keep it pressed to bring the auto-repeat into play. When you reach the end of the 4th line, where the cursor will stop and the machine will make rude beeping noises, press CTRL A and then ENTER. CTRL A gives 'ELSE' which will serve as our 'end of listing' marker.

We now have a TI BASIC statement which is 4 lines long - or do we ? CTRL U is the token for RANDOMIZE, and we have around 107 of them, so LIST and watch the longest TI BASIC line you'll ever see. (Well, almost: 'SEQUENTIAL' is one letter longer, but we can't get the token for it directly from the keyboard).

This is where life gets complicated. Type EDIT 1 and press ENTER. Eventually the screen will stop scrolling and the last word on screen will be ELSE, but where is the cursor ? ($64000 question). Answer: sitting over the 'R' in 'REM'. Where is that ? ($128000 question). Answer: I don't know, but try using FCTN D to move along the line (wherever it may be) and watch the screen scroll up one line for every character sideways that you move the cursor. Bring the auto-repeat into play and be prepared for an all-time-great crash (not audible). The psychedelic flashing, flickering colours and shapes you see are caused by the computer being forced to clog-dance through its own work area, upsetting the colours, 'patterns', even the Sprite descriptors. To get back to normality, exit from your 'editing' with FCTN 4 (BREAK) and then type RUN and ENTER. If all goes well, your normal screen colour should return and the shapes on screen (if they have been visible!) should resolve into normal letters etc.

I suspect that there may be a use for this. (You may suspect that I have a screw loose. I suspect that you may be right...)

If you possess a spirit of adventure, play around with this some more and let us all know what you find!

- -

M I C R O T I P S

Although the manual doesn't tell you so, the keys R, C, and E are 'active' when you OLD and SAVE to cassette. You can, for example, type SAVE CS1 and press ENTER, and then press C, and the computer will act as if you have already been through the full SAVE sequence and now want to check the program which has just been SAVEd. This has the advantage that you can check a program which you already have on cassette and compare it with the program which you have onboard to see if they are the same (perhaps to try and avoid making duplicate copies of an amended program). On the other hand, you may have wanted to SAVE the program but been called away to deal with a major catastrophe (the dog's been sick in the washing machine, your eldest has just taken the ears off the guinea pig with the Flymo, and the cat from next door has just dug up the gerbil you carefully laid to rest last week. And the week before. And the week before that...) in which case you can press E and exit, just as if you had encountered an error and decided not to pursue matters further.

This is one of those tips which can be of immense use to some owners and of no use whatsoever to others. I've always found it very useful: I have this habit of not checking a program with C having SAVEd it, and this enables me to check it later when I suddenly feel uncertain about the reliability of my cassette recorder. In addition I occasionally use CS2 to SAVE programs (usually because I've either gone and typed the wrong instruction without checking what I was doing, or because I've had trouble with my CS1 RECORD lead), and to check it I then change the leads round (well organised, you see), type SAVE CS1, press ENTER, and then press C, which allows me to check the program.

- -

Editor's note: There have been no entries for the Bulletin board and no response to Gary Harding's request for assistance. If any member feels that they could be of some help, please don't hesitate to get in touch with me.

- -

O N E S & T W O S C O M P L E M E N T

P e t e r B r o o k s

Don't worry if this leaves you with glazed eyes and gasping for air. It's a complex concept and I can't claim to be able to make the explanation easy. If you can't follow it at all, the solution is straightforward. Cough politely, and move on to the next article. Next month, try reading it again. And the month after that. Until suddenly, you'll find bits dropping into place. You may even read a far more clearly-explained account in the meantime; either way, the exposure to it here will have paved the way.

Although you may not think so now, sooner or later you're going to take an interest in programming in machine code - if you haven't done so already. When you reach that point, it can help a little if you have already been exposed to some of the concepts involved. In fact, if you have Extended BASIC, it can help to explain how NOT functions in statements like LET A = NOT B.

To begin with, we have to broach the thorny subject of the BINARY numbering system. We may be surrounded by binary systems (e.g., light switches: ON and OFF - or, like mine, ON, fizz-bang!ouch! and OFF) but we are so decimal-oriented that counting in tens seems the only way to count.

Until, that is, the day that number one son comes home, holds up ten fingers, and tells the family that he can count up to 1023 on them.

In the decimal or DENARY system, the digits 0 to 9 are used in combination to represent quantities. If you were brought up to think of numbers in this form:-

Thousands	Hundreds	Tens	Units	Tenths	Hundredths

or:-

1000	100	10	1	1/10	1/100

then life may be a little easier for you.

The decimal number 123 is really ONE hundred plus TWO tens plus THREE units (a UNIT is Maths jargon for ONE), or (1 x 100) + (2 x 10) + (3 x 1). In the case of the headings above, each column is the 'BASE' (10) times the value of the one to its right. Notice how it can be extended in both directions.

In the dreaded BINARY system, only the digits 0 and 1 are used in combination to represent quantities. The column headings are then:-

Thirty-twos	Sixteens	Eights	Fours	Twos	Units

or:-

32	16	8	4	2	1

Decimal 123 when expressed in binary terms is (1 x 64) + (1 x 32) +(1 x 16) + (1 x 8) + (0 x 4) + (1 x 2) + (1 x 1) or 1111011.

Generally, in computer terms eight binary digits (BITS) are used at a time - a BYTE - and this gives column headings of:-

128	64	32	16	8	4	2	1

The combinations of 0s and 1s range from 00000000 to 11111111, which, in decimal terms, runs from 0 to 255. (Work it out: 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1).

When it comes to treating these combinations in a byte as numbers and trying to perform a little maths with them (as you would when programming in machine code) a few difficulties arise. Firstly, for straightforward Maths you won't be able to count higher than 255 decimal unless you begin to use short machine code routines to handle several bytes at once (in the 99s case, there are special machine code commands which work on two bytes at a time, giving combinations of 0 to 65535.).

Secondly, you won't be able to handle fractions, nor will you be able to cope with negative numbers, so any subtractions like 4 - 6 will not be possible.

To get round this last problem, the use of a byte to represent a number is changed. Only 7 out of the 8 bits are used to represent a number and the 8th bit is used as a 'sign' bit - to indicate whether the other 7 bits represent a positive or negative number.

Corrected issue 3: 0 To -127

We can now count from 0 to +127 (00000000 to 01111111), and with the 8th bit set, we can count from 0 to -~~128~~ (00000000 to 11111111). However, life is never simple, and it is not as easy as that. To show why this particular method of indicating the sign won't work correctly, try adding 3 and -3. The result should be 0. Using the simplistic system just outlined, 3 will be 00000011, and -3 will be 10000011.

```
 00000011
 10000011 +
 10000110
```

The result is -6. If you have any difficulty deciding how the result was arrived at, drop me a line and I'll give a shortie in a later issue on simple Maths in binary.

So, how can we represent negative numbers accurately ? The answer is confusing and complicated. There are two steps involved. The first is 'inversion', or ONES COMPLEMENT (otherwise also known as NOT). What you do is to produce the positive version of the number which you want to make negative. To get -3, then, you begin with +3, which is 00000011 in binary. You then make all the 1s into 0s, and 0s into 1s. This will turn 00000011 into 11111100. Are you with me so far ?

Then you add 1 to the inverted bits. This turns 11111100 into 11111101, and to all intents and purposes 11111101 is -3. Note that the 8th bit is set to 1, signifying a negative number. This addition of 1 to Ones Complement is called TWOS COMPLEMENT and is a very important procedure.

To test this rather odd-looking method of obtaining a negative number, let us again try adding 3 to -3. What we now have is:-

```
  00000011
  11111101 +
 100000000
```

We have ended up with a 9 digit binary number. Har, har! I hear you say. So much for your Maths. However, bytes only come in groups of 8 bits, and the 9th bit is actually 'lost' - it is an 'overflow' - leaving us with 00000000, which is zero in any numbering system.

Twos Complement takes advantage of this overflow and subsequent loss of the 9th bit. And no, I don't know why they call it Twos Complement when there is no 2.

Having got this far, let us examine some other combinations. What, for example, does 11111111 represent ? The 8th bit indicates that it is a negative number, only which one ?

To discover this, we need to reverse the procedure involved in Twos Complement.

We must first subtract 1, and then invert the result. 11111111 - 1 = 11111110. Inverting this gives 00000001, or 1. Thus 11111111 is actually -1! If you are not convinced, try producing -1 using Twos Complement. Remember, get the positive form first: +1 is 00000001. Then invert (Ones Complement): 11111110. Then add 1: 11111111.

What then is the largest negative number that you can produce ? Earlier I said that using the first (and incorrect) method the largest value was -128. What does that look like in binary ? Take +128: 10000000. Invert: 01111111. Add 1: 10000000. That may have caused a double-take, but it is correct.

So far we have examined just a few numbers. Is there an easier method ? Well, in fact there is. If you look at the numbers again, see if you can spot the pattern:

BINARY NUMBER	UNSIGNED DECIMAL	SIGNED DECIMAL
00000011	3	+3
11111101	253	-3
11111111	255	-1
10000000	128	-128

The 'unsigned' values are those you get if you forget that the 8th bit is acting as the 'sign'. First, notice that if you forget the '+' and '-', the signed and unsigned values always add up to 256. Second, notice that with the negative values, if you subtract 256 from the unsigned decimal value, you obtain the signed value. Or, to put it more logically, the negative value is 256 minus the positive value. Thus -3 is 256 - 3 (=253), -1 is 256 - 1 (=255) and so on. No need to go through all the Twos Complement process. To reverse the process, you simply subtract the unsigned value from 256. (I know it sounds confusing, but keep at it!).

To check this, try obtaining the negative value for 103 using both methods.

Using Twos Complement, 103 in binary is 01100111; inversion gives 10011000, add 1 is 10011001 (which is 128 + 16 + 8 + 1, or 153), representing -103.

Using the last method, 256 - 103 is 153. Quick, eh ?

Having suffered all that, how does it help to explain how NOT works in Extended BASIC ? When used thus:

```
A = NOT B
```

NOT operates as an inverter, giving ONES COMPLEMENT. If Twos Complement gives the negative value of a number, and Ones Complement is 1 less than this, then NOT is equivalent to producing the negative number minus 1. Thus NOT B is the same as -B-1, the use for which escapes me for the moment...

- -

C L O S E   F I L E

That's all there is for this issue, I'm afraid. Until things stabilise, I will not be able to condense the material in each issue, or provide a greater variety of subjects. There is one piece of good news, however. TI may have withdrawn from the market, but in the wings there are a number of medium-sized firms waiting to put a number of products onto the market. The voice recognition unit has gone on sale in the USA, the quality of software is increasing all the time while the price is slowly becoming more realistic, and I understand that the current dearth of TI's own Extended BASIC module may be eased by an independent producer, PARCO ELECTRICS of Honiton in Devon. We are waiting for licensing agreements to be settled it seems, and then we could be in for a bright year in 1984.

Pete Brooks