1401 family programming manual

September 2012

First version	September 1995
Revised	January 1999
Revised	July 1999
Revised	July 2001
Revised	December 2004
Revised	June 2007
Revised	September 2007
Revised	January 2009
Revised	October 2010
Revised	August 2012
Revised	September 2012, removed Standard and *plus*

Published by:

Cambridge Electronic Design Limited
Science Park
Milton Road
Cambridge
CB4 0FE
UK

Telephone:	Cambridge (01223) 420186
Fax:	Cambridge (01223) 420488
Email:	info@ced.co.uk
Web:	http://www.ced.co.uk

Table of Contents

Table of commands

Introduction

Audience

This manual is intended for a programmer who has to write software to drive the micro1401, Micro1401 mk II, Micro1401-3, Power1401, Power1401 mk II or Power1401-3. There is an older version of this manual that also covers the standard 1401 and the 1401*plus*. If you use a 1401 through a turnkey software package (for example CED Signal or Spike2) and do not intend to write programs in C++ or Visual Basic to control a member of the 1401 family, it is unlikely that you will want to do more than flick through this manual, admiring the thickness, and giving thanks that others have done battle with it on your behalf.

Details of language support for your chosen operating system and language can be found in the *CED 1401 Family Language Support Manual*. This also provides more detailed information on communicating with the 1401 under your operating system.

Programmers who intend to write their own commands for the 1401 will need the 1401 Command Developer Pack for the micro1401, Power1401 or Micro1401. These packs contain the necessary tools to build 1401 commands for all members of the 1401 family. For the micro1401, Micro1401 mk II and the Power1401 you will also require an ARM developer toolkit from ARM limited. To develop for the Micro1401-3 or the Power1401 mk II or Power1401-3 you need the CrossWorks development kit from Rowley. CED also offers a command writing service; ask us for details.

Examples

The boxed programming examples in this manual show what is transmitted to the 1401 and when. They do not refer to any specific programming language or operating system. Working versions of the examples, numbered to correspond with this manual, are supplied on disk with the Language Support libraries.

Timing information

Throughout this manual we give timing information to help you to plan your application and determine what is possible to achieve and what is not. These timings were taken with real equipment in real programs and are our best effort to give a realistic measure of performance. However, unless you reproduce exactly the circumstances of our tests you will not get precisely the same results. Thus you should take the figures in this manual as a guide only. Further, they assume that the available processing power of the 1401 is not diluted by other activities (for example interrupts). Similarly, maximum ADC sample rates or DAC output rates assume that there are no competing tasks.

Older hardware

This manual is written for the latest releases of hardware and software; users of older units will find it interesting, but should be aware of the section at the end, covering differences they might find.

What is not in this manual

You will not find detailed hardware descriptions, troubleshooting guides and day-to-day maintenance issues within these pages. This book is thick enough already! The determined insomniac can find light relief in the companion tomes: *The micro1401 Owners handbook*, *The Power1401 Owners handbook*, *The 1401 Technical reference manual*, *The micro1401 technical manual*, *The Power1401 technical manual*, *The Micro1401 mk II technical manual*, *Writing commands for the micro1401* and *Writing commands for the Power1401 and Micro1401*. There are also circuit diagrams available for those who need them and are prepared to sign non-disclosure agreements. This documentation is available from CED.

About this manual

The 1401 family of interfaces

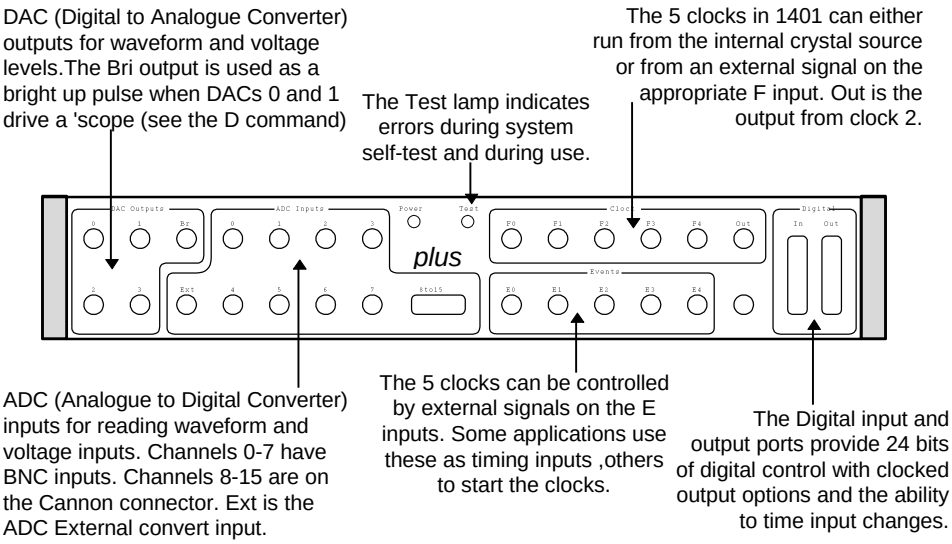
The 1401 family of interfaces are intelligent peripherals that generate and receive waveform, digital and timing signals. Using their own processors, clocks and memory, under the control of the host computer, they make complex real world jobs easy to control. There are ten family members: standard 1401, 1401*plus*, micro1401, Micro1401 mk II, Micro1401-3, Power1401, Power1401 625, Power1401 mk II, Power1401-3. We place a lot of emphasis on software compatibility; it is easy to write programs that can drive all the family members.

The standard 1401 and 1401*plus* are obsolete and are not covered in this manual, other than to note differences. There are older versions of this manual available from CED with programming details for these units – a few still survive more than 25 years after issue.

The standard 1401
first available in 1984
now obsolete

The standard 1401 had a 4 MHz processor, a 12 µs ADC (Analogue to Digital Converter) for sampling 16 channels of waveform data with a separate processor (the Z8 channel sequencer) for automatic channel changing and burst generation, 4 DACs (Digital to Analogue Converters) for waveform output, five clocks, event inputs, digital input and output with clock links and a memory space of around 60 kB for data and commands.

Front panel of 1401 and 1401plus



In addition, there were several option cards that included:

- MassRAM card for 2 or 8 MB of extra data storage memory and faster sampling
- Expansion of the 16 ADC channels to 32 channels
- Programmable 8 channel event detector
- Programmable gain and filter card options
- Fixed gain and filter cards

The *1401plus*
first available in 1991
now obsolete

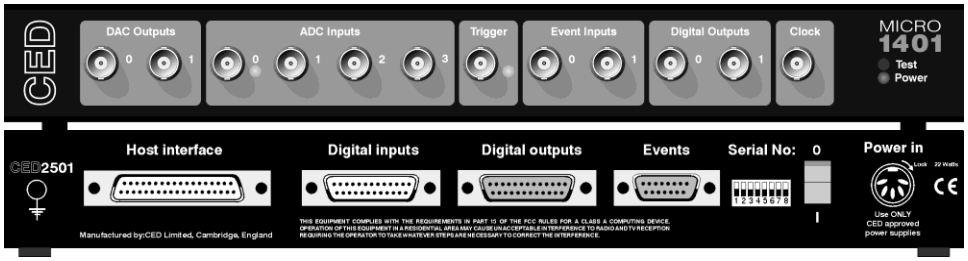
The 1401*plus* used a 20 MHz 32-bit processor for 20-40 times more processing power and increased the data space from 59 kB of the standard 1401 to more than 900 kB (16 MB with expanded memory). It was hardware compatible with the standard 1401.

The 1401*plus* supported the same options as the standard 1401 except for the MassRAM, which was emulated by a 1401*plus* with expanded memory. It used the same analogue card as the standard 1401 with the Z8 channel sequencer. However from 1993 it was fitted with a more advanced analogue card 'Issue-M' with a fast 3 µs ADC complete with ADC-silo and high-performance hardware sequencer. There were also analogue card options with 2.5 and 10 µs 16-bit ADC and 4 16-bit DACs for higher accuracy.

The micro1401
first available in 1996

The micro1401 has the speed and almost all the features of a 1401*plus* with the issue-M analogue card, packed into a much smaller space. Some sacrifices were made in the basic unit; there are only 4 ADC channels and 2 DAC channels as standard. However, there are benefits too: it is small and easily portable, all inputs have LED indicators to show when inputs or outputs are in use, interrupt driven commands generally run faster than 1401*plus*, trigger inputs (as seen by the user) are easier to understand and the unit can be expanded with more channels. It also has the option of a USB interface.

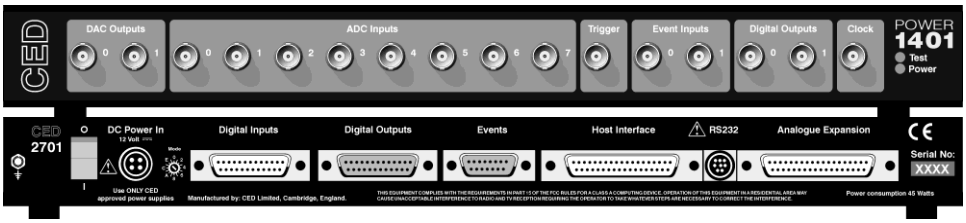
micro*1401*
front and rear panels



The Power1401
first available in 2000

The Power1401 takes the best features of the 1401*plus* and the micro1401 and adds a more powerful processor (up to 30 times faster than the micro1401 or the *plus*), a 16-bit analogue section and up to 256 MB of memory. Like the *plus*, it has 4 DAC channels and 16 ADC channels as standard, like the micro it has a small chassis and LED indicators. It also supports both the standard 1401 interface and USB 1. The software and hardware configuration is held in flash memory and can be updated without opening the unit. The Power1401 625 was a revision in 2004 with USB 2 and faster multi-channel sampling.

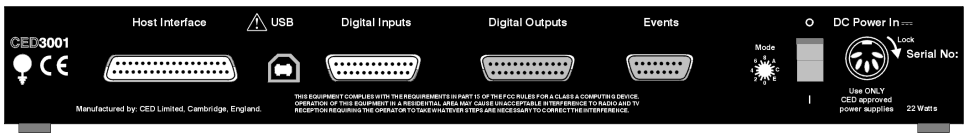
Power1401
front and rear panels



The Micro1401 mk II
first available in 2001

The Micro1401 mk II looks like the original micro1401 from the front, but it takes much of the internal structure from the Power1401. The processor is more than three times faster than the micro1401, it has a 16-bit 500 kHz ADC, a memory size of 1 or 2 MB and has firmware stored in flash memory for easy update without opening the box. It supports both the standard CED interface and USB.

Micro*1401 mk II*
frear panel



The Power1401 mk II
first available in 2007

This unit is similar to the Power1401, but with a processor some 3 times faster, up to 1 GB of memory and a faster multi-channel sample rate. It has a USB 2 interface.

The Micro1401-3
first available in 2009

The Micro1401-3 is very similar in appearance to the mk II, but has a faster processor, 4 MB of base memory and 16-bit DACs in place of the 12-bit DACs of the mk II. It has a USB 2 interface.

The Power1401-3
first available in 2012

This unit is similar to the Power1401 mk II, but with a faster processor, up to 2 GB of memory and a USB interface that has about twice the throughput.

Software compatibility

All members of the 1401 family use the same software interface. It is easy to write applications that will run with any 1401. The language support libraries are written to conceal differences between family members; however, applications that wish to take advantage of 1401-specific features are also supported. See the appendices at the end of this manual for differences you should be aware of.

Nomenclature

In this manual '1401' refers to all 1401 family types. To be specific we use 'micro1401', 'Micro1401 mk II', 'Micro1401-3', 'Power1401', 'Power1401 625', 'Power1401 mk II' and Power1401-3. As shorthand, we also use 'micro1', 'Micro2', 'Micro3', 'Power1', 'Power 625', 'Power2' and 'Power3'. Micro1401 means both mk II and -3, micro1401 (lower case) is the original.

Using the 1401

You control the 1401 by writing text strings to it, in the same way that you would write a string to a printer or to a file. For example, to read the values of ADC inputs 0 and 2:

```
ADC,0 2          This is the text string you send
96,-16           The 1401 sends back the result as text
```

This form of communication is fine when you wish to read an occasional value, or are setting up the 1401 to fill its internal memory with data. It would be rather slow for transferring 50 kB of data to host memory to write to disk.

The 1401 also supports a fast *block transfer* mode to move areas of memory directly between the 1401 and the host. With a PCI interface card, the micro1401 can transfer at around 1 MB per second, and the Micro1401 and Power1401 transfer at around 1.6 MB per second (limited by the host interface). The USB 2 interface allows much faster transfers, for example the Power3 can move data in excess of 40 MB per second.

The actual block transfer rate may be limited by the host computer and interface card (for example the (obsolete) 1401-10L interface combined with an IBM AT limits us to 250 kB per second). The highest performance interface is currently USB 2.

Writing programs to drive the 1401 is straightforward if you are familiar with almost any sort of programming. The remainder of this book describes the details.

Multi-tasking

Like 'dumb cards', a 1401 can be used simply, for example to capture 50,000 bytes of waveform data, sampled every 20 microseconds. It can also do several operations at once. By using multi-tasking commands (interrupt driven), the 1401 can simultaneously:

- Play a stimulus waveform on an analogue output
- Record a response waveform
- Record event times and reduce them to a time histogram
- Return data to the host for graphical display

Signal conditioning

Before considering application software at all, users will have thought about connecting transducers to the 1401. The most common signals are waveforms, for digitisation. If the frequency content of these waveforms is of interest, they must be low pass filtered to remove potential aliasing effects. See the chapter on FFT-related commands for a fuller description of this topic. CED has a range of suitable filters, and there are other sources.

The standard full scale input range of the waveform inputs is ±5 Volts with a resolution of around 0.16 mV (10 Volts / 65536). Input signals should be amplified so that the expected maximum amplitude, if possible, is between half and full scale, or potential resolution is lost. Again, CED or other amplifiers are suitable.

Some users, who are interested only in timing information from their signals, need to convert the waveforms to TTL-compatible pulses, for the event or digital input ports. This can be done with external discriminators; it is also possible to discriminate waveforms in software, for example by the `PERI32` command.

Memory

The table has the RAM memory in the 1401 family devices plus the expansion options. The operating system uses some of the space, but the majority of it can be used for data storage (for example, the micro uses about 100,000 bytes for the operating system, leaving 900,000 or so for data. At the other extreme, the Power3 reserves 3 MB for system use, leaving the rest for data.

1401 type	Base	Expansion
micro	1 MB	-
Micro2	1 MB	+1 MB
Micro3	4 MB	-
Power1/625	32 MB	to 256 MB
Power2	256 MB	to 1 GB
Power3	1 GB	to 2 GB

Memory expansion for Micro1401 mk II

The Micro1401 mk II has space on the motherboard for two sets of SRAM. Each set is 1 MB, allowing for 1 or 2 MB of fitted memory. The units have 1 MB as standard.

Power1401 memory sizes

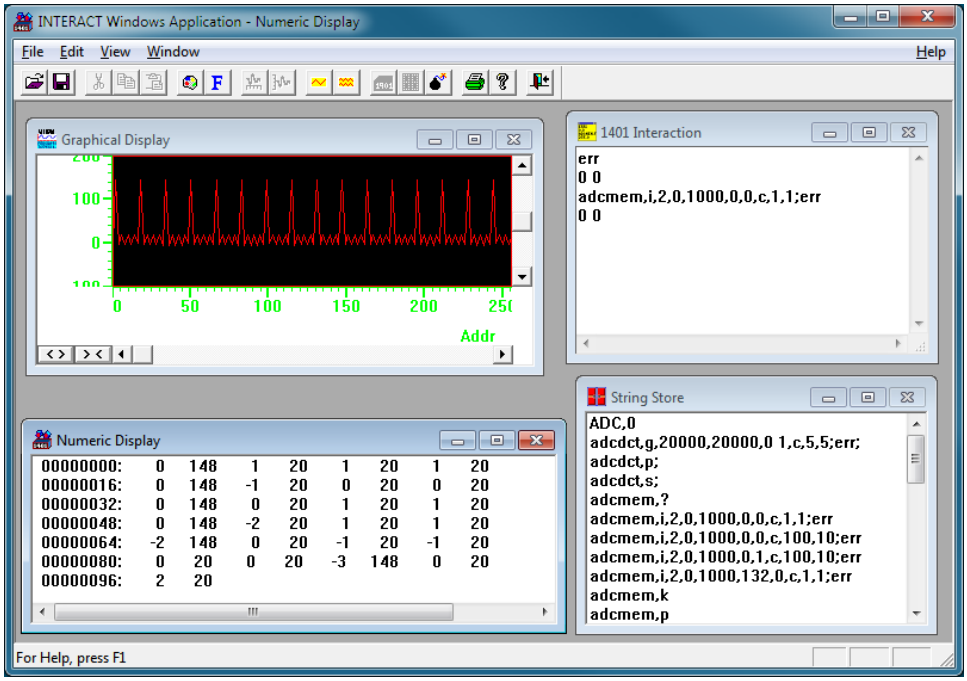
The Power1401 and 625 can be fitted with 16, 32, 64, 128 or 256 MB of memory. The Power1401 mk II can have 256 MB, 512 MB or 1 GB of memory. The Power1401-3 has 1 or 2 GB.

Memory expansion for Micro1401-3

The Micro1401-3 has space on the motherboard for two sets of SRAM. Each set is 1/2 MB, allowing for 1-4 MB of fitted memory. The units have 4 MB as standard.

The interactive programming tool INTERACT

The INTERACT program is an interactive tool to help with the development of 1401 applications and commands. It replaces the host side of the dialogue with you, typing on the keyboard. The Windows version is called Interw32.exe. We suggest that you make use of this program while you learn to program your 1401. INTERACT allows you to load and try out the commands described in this manual.



The full features include:

- Loading of command code to 1401 from disk.
- Display of revision levels of loaded commands and of the Monitor ROM. The Monitor level shows as the revision of the RESET command.
- Interactive running of commands from the host keyboard
- Building, saving and running of command sequences
- Graphical display of 1401 memory either in static form or with continuous updates for use with 'multi-tasking' or 'interrupt-driven' commands

The best way to learn the program is simply to try it. There are four windows that you can display:

Graphical Display This allows you to display a range of 1401 user memory as 8-bit, 16-bit or 32-bit values and as signed or unsigned values.

Numeric Display This is the same as the Graphical display except the values are displayed as number, either as decimal or hexadecimal.

1401 Interaction You can type commands in this window and see the 1401 responses. You can copy commands from this window to the String Store.

String Store This window holds a list of 1401 commands that you can run by double-clicking.

The menu system holds commands for configuring the windows and to copy data between the interaction window and the string store. There is also a toolbar with short-cuts to the most commonly used commands.

INTERACT as a learning tool

Many users find INTERACT very useful as a learning aid when they begin programming the 1401; we also use INTERACT to try out command sequences before we build them into a program and as a test-bed for new 1401 commands.

Commands to avoid!

Any command may be typed, but unwelcome effects may result from use of a small group. These are `RESET`, and the commands that exchange data blocks with the host computer: `TO1401`, `TOHOST` and some specialised advanced data capture commands.

The `RESET` command clears the 1401 as at power up, separating it from the host communication. The 1401 can be reconnected by pressing the function key for Reset.

Writing programs for the 1401

This chapter introduces you to programming with the 1401. It does not aspire to teach you to write programs on your host computer; if you are new to programming, you must get that instruction somewhere else! It gives an overview of 1401 use and a description of the general low level communication commands. The rest of the book covers the more specialised commands.

The 1401 is driven by sending text strings; in principle, any language that can drive a printer is suitable. We have prepared libraries to simplify the use of 1401 from several popular languages, see *The Language Support Manual* for more details.

Fundamentals of 1401 use

Programs running on the host computer send the 1401 instructions as text strings. The first few characters of the string contain the 1401 command name, the rest of the string contains additional parameters needed to specify the command. The 1401 Monitor software interprets this string and invokes the appropriate command. If the requested command is not present in the 1401 or a command parameter is incorrect or there is a problem during the command operation the 1401 monitor turns on the Test light and records an error. Errors can be read and cleared by the `ERR` command.

Results can be sent back to the host as text strings. Each string is a list of decimal numbers separated by commas and terminated by a carriage return character. However, most high-level language users retrieve values from the 1401 using a library function to read the values and convert them from text to numbers; you are referred to the documentation for your specific language for more details.

Some commands are 'built-in' in the 1401 Monitor ROM/flash memory, but most are loaded by the user from disk at the start of the session. All commands should be assumed to need loading unless the description says that they are built-in. The Language Support libraries provide easy-to-use routines for loading commands from disk.

The command operation might result in the storage of some data in the 1401 memory, initiation of an interrupt driven process that will continue after the command operation, processing of some data in 1401 memory or the return of information to the host computer. Once the command operation has finished, the 1401 is ready to interpret and execute the next command.

1401 text buffers

When a text string is sent to the 1401 it is saved in an input buffer if the 1401 is not ready to deal with it immediately. This buffer will accept data, even while the 1401 is busy running another command. Up to 255 characters can be stored in the input buffer (enough space for 5 to 10 typical commands). Similarly, when the 1401 outputs text strings to the host they can also be stored in a 255 character output buffer if the host is not ready to accept the data.

These buffers help to increase system performance; neither the host computer nor the 1401 need ever wait for the other to be ready (unless a buffer becomes completely full).

The buffers are completely transparent to the programmer in normal operation, however, users can take advantage of them by writing a series of text commands to the 1401 and leaving the 1401 to process them while the host gets on with other tasks.

Synchronisation between the host and the 1401

The 1401 usually starts executing a command some time (typically a few milliseconds) after the command string has been sent. If another command was in progress at the time, the delay could stretch into seconds. However, the 1401 executes commands in the order that they were sent, and never starts a new command until the previous one has been finished. Interrupt driven processes and completion routines can continue to run in the background while commands are processed.

Because of this there is rarely any need to worry about synchronisation. If the 1401 command returns results to the host, the host program can wait to read the results, and know that at that point the command, and any sent previously, have been processed.

Types of commands

There are two basic types of 1401 command: *sequential* and *multi-tasking*. When the 1401 receives a text string that activates a sequential command, the 1401 will complete all operations associated with the command before it can process another text string. In contrast, when a text string requesting a multi-tasking command is received, the 1401 starts the requested operation, then continues to process text strings from the host while the multi-tasking command continues to run.

Multi-tasking is achieved using hardware interrupts in the 1401. When a device in the 1401 needs attention, for example when the ADC used to read waveform data values has a new value ready, it requests an interrupt. The 1401 stops what it is doing, saves the current state of the system, then branches to a special routine (the interrupt service routine) within the multi-tasking command. The interrupt service routine attends to the hardware, (for the ADC it reads the value and saves it in memory) then restores the state of the system so it can continue with the interrupted task.

More than one interrupt driven process can be used at a time, the only restriction being that each must be driven by a separate interrupt source. Simultaneous interrupts are prioritised by hardware built into the 1401. If the interrupt rate is too high, some interrupts may be missed. However, the 1401 has special hardware to detect that interrupts have been lost so there is no danger of this happening without warning.

The 1401 family has been designed to handle interrupts quickly. A typical interrupt service routine takes around 3 µs in a micro and a microsecond or so in a Power1/2 or Micro2/3. The Power3 is capable of dealing with more than one interrupt per microsecond. The table shows suggested maximum total interrupt rates that you can achieve with each 1401.

1401	kHz
micro1	250
Micro2	>400
Micro3	>500
Power1/2	>500
Power3	>1000

Sequential commands

Typical of these are the array arithmetic commands. The sequence of actions when a sequential command is used would be:

1. Host sends command string to 1401.
2. Monitor interprets command and checks parameters.
3. Command is executed, and does whatever it must.
4. Command sends back results as text, if required.
5. Monitor moves on to the next command.

Sequential commands can run in the background while an interrupt driven command also runs. Because of this, you must be aware of possible contention for resources. For example, if the ADC is in use for a multi-tasking command (for example `ADCMEM`) it cannot be used by the sequential `ADC` command.

Multi-tasking commands

Multi-tasking commands have a sub-command that initiates an interrupt driven process and then quits, leaving the interrupts to continue. Other sub-commands will allow the state of the interrupt driven process to be monitored or will stop the interrupts. The interrupt driven process can also stop further interrupts if necessary. The sequence of actions for this type of command is usually:

1. Host sends command string to 1401.
2. Monitor interprets command and checks parameters.
3. Command is executed, and initiates the interrupts.
4. Command returns control to Monitor leaving interrupt process running.
5. The normal sequence of command transmission and execution continues for an indefinite period. Normal commands can be used and other interrupt driven processes may be started. Throughout all this, the interrupt driven process continues.
6. Either by means of a sub-command, or by the interrupt driven process itself, the interrupts are stopped and this process terminates.

Commands of this type, which are all loadable, include interrupt driven waveform capture (`ADCMEM`), waveform output (`MEMDAC`), pulse time logging (`AUDAT`) and digital sequencing (`DIGTIM`).

Completion routines

This is the term used to describe a limited extra form of multi-tasking. Completion routines run once for each command the 1401 executes and whenever the 1401 is waiting for a command to be sent and thus has nothing else to do. This is what the 1401 spends the bulk of its time doing, so a completion routine, once set up, executes more or less continuously, with gaps whenever a command is executed.

For example, the code that maintains the LEDs on the front of the micro and Power1401s is a completion routine that asks each 1401 command the resources that it is using and turns LEDs on and off as required.

Completion routines are commonly used by advanced interrupt driven data capture routines to transfer blocks of data between the 1401 and host computer. Spike2 makes extensive use of completion routines. More information is available in the command writing kits for each 1401.

Use of 1401 memory

Many 1401 commands need an area of 1401 memory in which to work. The `ADCMEM` command, for instance, fills an area of memory with ADC data; the `MEMDAC` command outputs the data in an area of memory to the DACs and the `FFT` command transforms the data stored in an area of memory in place.

We specify the memory in bytes. Each byte of information has 8 bits and can take up to 256 values. Most 1401 commands make use of 16-bit (2 byte) data but both 1 byte and 4 byte data are also used. The ranges of all the types are:

bytes	bits	unsigned range	signed range
1	8	0...255	-128...127
2	16	0...65535	-32768...32767
4	32	0...4,294,967,295	-2,147,483,648...2,147,483,647

Where more than one byte is used to represent a value, the bytes of lower value are stored at the lower addresses (little-endian). This matches the storage protocol used by almost all modern PCs. This only matters when direct block transfers of memory between the 1401 and host are required. Where data is transferred as text there is no difference.

Specifying memory

All commands that make use of 1401 memory specify it as a start address and a size. Both the start and the size are specified in bytes.

Which area of memory to use is entirely up to the programmer; there are no areas of user memory that must be used for a given purpose. The 1401 software checks the area of memory specified to see if any of it lies outside the user area, and generates an error if it does. The `MEMTOP` command reports the available memory in a 1401.

We recommend that at least a 32-bit integer type (`LONGINT`, `INTEGER4`, `INTEGER*4`, `int32_t`) is used to store 1401 addresses. At the time of writing, no user space exceeds 2 GB, so there is no problem with 32-bit integers wrapping around and becoming negative. It would be better to use an unsigned type (where these are available in your language).

The 1401 doesn't check for areas of memory used by different commands overlapping. It is the responsibility of the user or applications programmer to ensure that this does not happen in such a way as to cause corruption of data. It is perfectly reasonable, and very common, for two commands to use the same memory in turn, for instance first to collect ADC data and then to process the data. The problem that needs to be watched for is a command overwriting data that is being stored for later use.

Format of 1401 commands

An instruction is sent to 1401 as a text string: a list of ASCII characters terminated by a semicolon or Carriage Return (ASCII code 13). The instruction is split into fields by commas. There are two types of field: *numeric fields* hold a number or a list of numbers separated by spaces, *character fields* hold one or more characters. The number of fields and their content, depends upon the command. Some typical 1401 instructions are:

```
ERR;DAC,0,1024;ADC,0 1 2 3,2;
```

The first instruction has one character field: the text string `ERR`. The second has three fields: a character field `DAC` and two numeric fields `0` and `1024`. The third also has three: a character field `ADC`, a numeric field with a list of four numbers `0 1 2 3` and a numeric field holding the number `2`.

The first field of an instruction holds the command name (up to 7 characters). If this is incorrect or no command in the 1401 matches the name then the result is always an error code of 255 and the Test indicator is turned on (turns red in modern 1401s). The remaining fields in the instruction string are command parameters; information specifying the exact action wanted from the command. The arrangement of parameters often follows common conventions, but this is not mandatory.

The second field (the first parameter), is often a character field indicating the action required from the command; `?` to query the state of the command for instance, or `K` to kill a multi-tasking command. The number of fields required and their meaning will vary according to this second field, often called the sub-command character or specifier.

Conventions used in this manual

Throughout this book, wherever command syntax is shown, fields in `[square]` brackets are optional, and items at the end of a command example, after a semicolon, represent values returned by the command to the host. For example:

```
ADC,chan[,byte];values
```

In this case the field named `byte` can be omitted and the command returns data. Field names used in the text, such as `chan` and `byte`, are shown in a different font.

Character fields

The most common example of a character field is the first field of a command, which is the command name. Character fields have a minimum and maximum number of characters that will be accepted and all characters (including spaces) are significant.

The 1401 normally converts lower case characters to upper case. You can force lower case to be preserved in all 1401s except the standard by surrounding character fields in double quote marks. We show 1401 commands in upper case in our manuals.

Numeric fields

Numeric fields may hold one or more numeric expressions separated by spaces. Numeric expressions are composed of decimal numbers, hexadecimal numbers, local variables and operators. All numbers as 32 bit quantities. The maximum range of numbers is from -2,147,483,648 to 4,294,967,295. Whether a number is treated as signed or unsigned depends on the command and the field in the command.

Common errors with numeric fields

Commands define the range of acceptable numbers for all their fields and if a field is a signed or unsigned number. The numeric range for a field is usually smaller than the maximum range possible. The most common error users make with numeric fields is to send the name of a variable in their program to the 1401 when they meant to send the value of the variable! The second most common error is to send a decimal point. 1401s only deal in integral values; a decimal point will always cause an error.

Decimal numbers

A decimal number is composed of the characters 0 to 9. The command sets the acceptable range of the number. Negative numbers have a leading minus sign.

Hexadecimal numbers

A hexadecimal number is composed of the `$` (dollar) character followed by the characters `0` to `9` and `A` to `F` or `a` to `f`. The number is treated as a bit pattern and its use as signed or unsigned is determined by the command.

Local variables

There are 26 local variables defined in 1401 named `A` to `Z` in upper or lower case. These variables can be used anywhere a numeric expression is expected. The variables hold 16 bit numbers in the standard 1401 and 32 bit numbers in the rest. See `VAR` and `RUNCMD` for details of saving values in the local variables.

1401 numeric operators

The 1401 allows brackets in numeric expressions and accepts a range of numeric operators. Operators are given a priority to allow expressions to be evaluated in a natural way. Unary operators have the highest priority and bind to their right. The following unary operators are allowed:

Unary operators

Operator	Use
-	negate the expression to the right
~	bitwise NOT of the expression to the right
@	Contents of the byte at user address to the right
!	16 bit integer at user address to the right
#	32 bit integer at user address to the right

The remainder of the operators are evaluated in order of priority, higher priorities are evaluated before lower priorities. Operators at the same level are evaluated from left to right. Logical values are 0 for false and not-zero for true. Logical results of comparisons are returned as 1 for true. Missing priorities are for possible expansion. The operators and priorities are based on the C language; higher priorities take precedence over lower priorities.

Binary operators

Operators	Priority	Use
* / %	13	Multiply, divide and modulo
+ -	12	Add and subtract
> >= <= <	10	Comparisons (signed)
== !=	9	Equal and not equal
&	8	Bitwise AND
^	7	Bitwise exclusive OR
|	6	Bitwise OR
&&	5	Logical AND
||	4	Logical OR

Expressions are evaluated to 32 bits with no warning of overflow. All numbers are treated as signed for calculation; the 32 bit numeric range should not make this a problem. The following are acceptable expressions that all evaluate to 100 (local variable `X` is assumed to hold 50):

```
100          (100*1)      1+2+3+47*2     (4>2)*100     4>2*100+100
X+X            X*2        (X-1)/49+99    100+1000%X     97+(3&15)
```

Data transfer formats

Data can be transferred between the host and the 1401 in two ways: either as textual transfers of ASCII strings holding numbers, or as block transfers of areas of memory directly from one system to the other.

Textual transfers

Textual transfers of data are the easiest to use. An example of this type of data transfer would be the numerical items in a command string.

When a command returns numbers these are formatted as a list of decimal numbers separated by commas with no embedded spaces, and terminated by the carriage return character. The host interface software for certain languages (Pascal, for example) may translate the commas into spaces to make the numbers easier to interpret. The possible range of returned numbers is -2,147,483,648 to 4,294,967,295.

Block transfers

Binary data transfers are more difficult to use, but they are invaluable when large amounts of data are to be transferred quickly, typically 100 to 200 times faster. The process amounts to copying one area of memory to another but it can give problems due to number storage formats and synchronisation. The process is greatly simplified if you are using a CED-supported language where we have support for block transfers.

Basic commands

The remainder of this chapter describes the basic set of commands that you almost always need to use in a program. Some of these commands, such as `CLOAD`, `TOHOST` and `TO1401` you may never use directly, but they will be used for you by your 1401 interface library for your chosen high-level language. Others, such as `ERR`, are part of the daily life of the 1401 programmer.

Boxed examples refer to example code provided on the 1401 language support disk.

CLIST
List commands

The built-in CLIST command reads and reports the names and revision levels of all commands in the 1401. Built-in commands are listed with loaded commands, with no distinction. Each name is terminated by a comma, followed by the revision level as x.y where x is the system level (20 for micro1, 30 for the Power1, 40 for the Micro2, 50 for the Power2, 60 for the Micro3, 70 for the Power3) and y is the level of the command revision, in the range 0 to 255. The revision level is terminated by a Return character (ASCII code 13), and the list of names ends with a line holding just a comma and Return. To use, just send the string CLIST and read the response:

Example 1: List available commands

Set up communication with the 1401
Send this string:
    CLIST
Set up a loop to read the response from the 1401, until empty, and print:
    command + revision level, on the screen.

A simpler direct way of checking whether a specific command is loaded in the 1401 is simply to send the command name and check the error state. There will be one of two errors: 255 or 254.

255 the named command is not recognised, so it is not already loaded.

254 the command is loaded; the name was matched but the number of argument fields was wrong, as no loaded command can be used without arguments.

CLOAD
Load a new command

A few of the 1401 commands are built-in; they are in the Monitor ROM and can be used immediately, but most of the commands have to be loaded from disk. The advantage of this system is that we do not have to waste space on commands that we do not need and it allows the capability of the 1401 to be extended. Commands are loaded using the CLOAD command and can be removed using KILL. Both these commands are built-in.

The language support libraries provide routines to automate command loading for you; you should never need to use CLOAD directly. We describe it here for completeness.

The first step is to determine if the command is already loaded in 1401. Techniques for this are described in the CLIST description above. If a required command is not present in 1401 the second step is to load the command into host memory from disk. You will find detailed code for this in the source code of the language support library. If you are writing your own support for a new language you will need the *1401 Technical Support* manual for details of the disk formats used. Finally, the CLOAD command is used to transfer the command image from host memory to the 1401 and to link it into the available commands. The general form for the CLOAD command is:

```
CLOAD,hoOff,size
```

hoOff is a number, which indicates the address in the host memory where the command to be loaded into 1401 starts, see TOHOST on page 17.

size is also a number, the size of the command in bytes.

Error codes are 253,0 if the data passed is not a command and 253,1 if there is not enough command space memory.

KILL
Unload commands

If memory space in the 1401 is at a premium, unwanted commands can be removed, one by one, or in a complete clear. The built-in command KILL removes one or more loaded commands from 1401 memory. The command has two forms:

KILL	Remove all loaded commands
KILL,n	Remove the last n (n < 256) loaded commands

The KILL command also performs an implicit CLEAR, cancelling all current operations before commands are deleted.

CLEAR
Initialise commands

The built-in CLEAR command cancels all 1401 operations and resets the system without deleting loaded commands. The set up phase of most commands will reset the relevant sections of the 1401 but it is good practice to run CLEAR or KILL when starting a new program. The CLOAD command does an implicit CLEAR both before and after loading a new command, so an explicit CLEAR is not needed if new commands are loaded. Characters in the buffer between 1401 and the host are not removed by CLEAR. The command has no arguments so it is used as:

```
CLEAR
```

RESET
Resetting the 1401 as at power up

The built-in RESET command makes the 1401 perform the same internal operations as it would on power up, which can take a long time if you have a lot of memory and the unit is set to do a full memory test. Only the built-in commands remain; all the loaded commands are removed. You should not usually need to use this command, except when you suspect that something terrible has happened to software running inside the 1401! The command has no arguments, so it is simply used as:

```
RESET
```

If the RESET command is used, the software interface will need setting up again, and note that any characters after the R E S E T and before initialising and re-directing the output will be lost. The setting up can be done by the Reset1401 language support library routine.

INFO
System information

The built-in INFO command is used by CED system programmers to obtain low-level information and to adjust the micro1401 ADC gain and offset trim. It has two forms:

INFO,S,n;value	Return system information for item n
INFO,T,gain,offset	Set ADC trimming values for micro1401 only

n — The system item to collect information on. Useful values are: 17 for the ADC Silo size, 19 for the channel sequencer hardware revision, 20 for the ADC block type (as returned by ADCBST,Z), 21 for the number of ADC channels, 22 for the number of available DAC channels, 24 for the maximum number of channels allowed in sequential mode for the ADC channel sequencer. Items 21 and 22 are supported by the Power1401 and Micro1401 and return 0 on earlier 1401s.

gain — The gain trim in the range 0-255. 128 is the centre value.

offset — The offset trim in the range 0-255. 128 is the centre value.

ERR
Check 1401 for errors

Requests for performance that the 1401 cannot achieve and faults in command syntax, are errors. These are stored and should be tested, particularly during program development. An exception to this is the fault of attempting to convert too quickly with the ADC in interrupt commands when the error is shown in the command status (see ADCMEM). Each error over-writes the preceding contents of the error register and turns on the red Test light on the front of 1401. The built-in ERR command reads the state of the error register:

```
ERR;code,qualifier
```

The ERR command returns an error code followed by a qualifier, shown as x in the table. If no explanation is given for x in the table, the value of x is undefined.

Error	Meaning of the error code
255,x	The command name given is unknown to the 1401.
254,x	There is an error in the argument list. The value of x is usually the field number in which the error was detected times 16.
253,x	A run time error occurred. x depends on the command. If the error was in the value of a field then x is often the field number times 16. Some commands define x; see the command documentation.
252,x	The expression evaluator detected an error in a numeric field.
251,x	Division by zero attempted during the evaluation of an expression.
250,x	An unknown symbol was found in an expression.
249,x	A command passed to the 1401 was too long.
248,x	End of line (CR character) in a string field introduced by ".
247,x	A memory reference was outside the user memory area.
31,0	Clock or ADC interrupt overrun (command error or running too fast)
17-30,x	Unexpected IRQ vector 0-14, x is the offending CSR value
1-15,x	Unexpected FIQ vector 0-14, x is the offending CSR value

Codes 1,x to 30,x are caused by unexpected interrupts due to hardware failures or bugs in commands. x is data about the interrupt source, usually the value of the CSR (Control and Status Register) causing the interrupt.

Example 2: Checking 1401 for errors

Set up communication with the 1401
Send this string and read the two responses:

```
ERR;E1,E2
```

Print the results on the host screen as:

```
"Error flag 1 = (E1) and flag 2 = (E2)"
```

RDADR
Read a 1401 data location

The built-in RDADR command reads small amounts of data from the 1401 to variables in the host. The data is transferred as text. The general form of the command is:

```
RDADR,byte,st;value
```

byte is a character that sets whether byte, word or long data is read. Character 1 sets bytes (range 0 to 255, unsigned) and 2 sets (signed) 16-bit words. 4 sets signed 32-bit numbers (not for standard 1401).

st is a number, the user area address in 1401 of the start of the data to be read.

```
RDADR,1,0;123          read location 0, holds 123
RDADR,2,1024;-23452    read locations 1024-1025, holds -23452
RDADR,4,100;123456     read locations 100-103, holds 123456 (not standard 1401 )
```

WRADR Write to a 1401 data location

The built-in WRADR command is the logical complement of RDADR and sets values in the user data space of the 1401. The general form is:

```
WRADR,byte,st,value
```

byte — This is a single character which is 1, 2 or 4 (not standard 1401) to indicate that the data to be written is 1 byte, 2 bytes or 4 bytes in size.

st — The start address in the user data space of the data to be changed.

value — The value to be written to the user data space.

```
WRADR,1,0,123;          set location 0 in user memory to 123
WRADR,2,1024,-23452;    set locations 1024,1025 in user memory to -23452
WRADR,4,100,1234567;    set locations 100-103 (not standard 1401)
```

These two commands are intended for transferring a few bytes of data; the built-in TO1401 and TOHOST commands are faster for large transfers, but need the absolute addresses of the arrays in the host machine.

TO1401 and TOHOST Block transfers of data

The TO1401 and TOHOST commands transfer a block of memory between the host and the 1401 user data space at high speed.

The *1401 Language Support* library provides routines that simplify the use of these two commands. You must consult the documentation of this library for more information on the use of block transfers on your computer type and operating system. In most cases, you will not need to program block transfers yourself.

These commands introduce the concept of an offset into the area of memory to be used in the host. For some hosts, this offset is the physical address of the host memory. However, for most hosts it is an offset from a particular position in the host memory map. Some hosts allow several areas to be defined simultaneously for transfers. Some advanced commands allow you to specify the target area for a transfer; this is described in the documentation for these commands. All commands in this manual use area 0.

Command variants

```
TO1401,st,sz,hoOff[,R]
TOHOST,st,sz,hoOff[,R]
```

st — is the start address of the block in the user data area of the 1401.

sz — is the number of bytes to be transferred.

hoOff — is the offset in the host (within area 0) to the memory for the data.

R — This is an optional character that causes each pair of bytes transferred to be swapped and is to support computers that use a big-endian byte order.

With most computers this data transfer uses DMA (Direct Memory Access) and will not slow down the host computer appreciably. If DMA is not available, the host computer will appear to stop during the transfer. An example using TOHOST appears in the *Voltages and Waveforms* chapter, in the ADCMEM section, and is included with the language support.

MEMTOP Memory information and control

The built-in `MEMTOP` command gets details of the memory space in the 1401 and allows control over the memory arrangement. The memory configuration of the 1401 is set on power-up.

MEMTOP variants

The `MEMTOP` command variants return the sizes of the memory areas, and configure these areas to suit an application.

`MEMTOP;top,base`	standard 1401 compatible command version; do not use
`MEMTOP,?;usrsz`	get user space
`MEMTOP,B;cmdsz,hsz,stksz,busz`	get base memory data

Compatibility variant obsolete

`MEMTOP;top,base` was the only `MEMTOP` command available on the standard 1401. The difference of these two numbers was the size of the standard 1401 user data space. This is only here to allow ancient software to work; do not use.

Get user area size

`MEMTOP,?;usrsz` returns the size of the user data area. The number returned can be very large so read it into a 32-bit (unsigned) integer type or a real number.

Get base memory information

`MEMTOP,B;cmdsz,hsz,stksz,busz` gets 1401 memory organisation information. The returned values are sizes in bytes.

`cmdsz` The size of the area allocated for loadable commands.

`hsz` The size of the heap. Some 1401s use the heap for loaded commands.

`stksz` The size of the stack area used for temporary variables and argument passing.

`busz` This is the size of the user data area.

ROM Control of additional ROM space

The micro1401 has two ROM (Read Only Memory) sockets. Socket 0 holds the monitor ROM, and 1 is normally empty. The monitor ROM holds the self-test code and the built-in commands. It is possible to fit an extra ROM, and to utilise unused space in the monitor ROM. The ROM command obtains information about the contents of these extra ROM areas and loads commands from these additional ROM areas.

The Power1401 and Micro1401 have a flash ROM with multiple areas that can be programmed separately. This memory holds the operating software and programming information for the configurable parts of the hardware. The major benefit of this flash memory is that it can be updated by software without any need to disassemble the 1401.

There are special commands for the Power and Micro1401. You can use the micro1401 variants, however the response will indicate that no ROM was found. We may choose to emulate some of the micro1401 features in the future.

Command variants for micro1401

`ROM,?n;format,revfmt,revdat,rflags`	Get ROM type
`ROM,Cn;text`	Return text describing ROM
`ROM,Nn;name,itype,iflags`	List contents of a ROM
`ROM,L,name`	Load command from ROM

`n` The ROM number; ROM 0 is the monitor ROM, 1 is the spare ROM position.

`format` The format of the ROM. -1 = ROM not found, 0 = ROM holding commands.

`revfmt` The revision level of the ROM format (currently 0), or -1 if no ROM found.

`revdat` The revision level of the data in the ROM in the range 0 to 99, or -1 if no ROM found. Values 0 to 9 should be interpreted as 00 to 09.

`rflags` -1 if no ROM is fitted , otherwise 8 flag bits of which only bit 0 is currently defined. Bit 0 is set if the ROM contains data to be loaded on powers up.

`text` A text string of up to 80 characters that describes the ROM.

`name` A text string with no embedded spaces to identify a command in the ROM.

`itype` The type of a data item held in the ROM. Type 0 is a 1401 command.

`iflags` Bit 0 of this value is set if this item is to be loaded into the 1401 on power-up.

Report ROM type

The `ROM,?n` command returns 4 numbers that are all -1 if the ROM space does not exist, or that return information on the ROM type. The main use of this command is to determine if a ROM is present. The `ROM,Cn` command returns a text string, terminated by end of line, which describes the ROM. This string can be up to 80 characters long.

List ROM contents

Each ROM can hold several data items identified by names. These names are up to 15 characters in length and can hold any printing characters in upper or lower case. By convention, only alphanumeric characters are used and names are upper case. A name need not be unique, but it is only possible to load the first data item that matches a name.

The `ROM,Nn` command is used to list the contents of ROM n. The command lists each item on a line, and terminates the list with a blank line. The command lists the type and the flags for each item in addition to the name. These three fields are separated by commas. For example, a ROM holding the `SS2`, `ADCMEM` and `MEMDAC` commands, with `SS2` and `ADCMEM` marked to be loaded on power-up might return:

```
SS2,0,1<end of line>
ADCMEM,0,1<end of line>
MEMDAC,0,0<end of line>
<end of line>
```

Load from ROM

Commands can be loaded from the ROM using the `ROM,L,name` command variant. This searches ROM 1 first, then ROM 0, looking for a data item of type 0 (a command) which matches name. If the name used to identify the command in the ROM contains lower case characters you must use lower case in name and enclose it in double quotes, otherwise the lower case is converted to upper case and will not match. Error 253,0 is returned if a command of the correct name is not found in the ROM. Error 253,1 is returned if there is not enough room in the loadable command area.

Command variants for Micro1401 and Power1401

We advise extreme caution be exercised before you write code to modify the contents of the flash memory. CED provides tools that can safely modify flash areas (for example in the Try1401 Windows program). None of these commands exist for the micro1401.

`ROM,F,R,image,st;sz`	Read flash contents into memory
`ROM,F,P,st,sz;result`	Program flash contents from block in memory
`ROM,F,I,image;sz,name`	Return information about a flash image
`ROM,F,C,image`	Clear an alternate flash image slot (will not touch factory images)
`ROM,F,V;cpld,fpga,cpldA,flgaA`	Return CPLD and FPGA firmware revisions

`image` The flash image to use. Valid image numbers are 0 to 15 (0 to 9 for the Micro1401. Image 0 is special and may not be visible to the ROM command.

`st` The start of a region in user memory space aligned to a 4 byte boundary.

`sz` The number of bytes in a flash image. A return value of 0 means that the flash image did not hold a recognised image (image 0 may not be recognised, even though it holds a correct image).

`result` A code to indicate how the program operation fared. A value of 0 means the operation was a success. -1 means that the image header was incorrect, -2 means that the image was corrupt (checksum failure).

`name` This is a returned text string that is either 16 characters long (padded on the end with spaces) or empty if the flash image is not valid.

Read flash contents

You can copy a flash image into 1401 user memory space with the `ROM,F,R,image,st` command as long as the image has a valid header and the image checksum is correct. The command returns the number of bytes in the image or 0 if the image does not appear to contain a correctly formatted image (or image 0 is requested).

Program flash contents

The `ROM,F,P,st,sz` command copies a memory image defined by `st,sz` to the flash. The flash block number to program is held within the image, which must be correctly formatted and have a correct checksum. Any flash block can be programmed by this command. The programming operation can take several seconds, so programs must wait for a response. A faulty flash memory could cause this operation to hang the 1401.

Return flash information

The `ROM,F,I,image` command returns information about a flash memory image. The returned values are a number followed by a comma, then either 16 characters or if the number was 0, then no characters. The number is the size of the flash image in bytes, and the characters are an identifying message built into the image. If there is no image in the flash area, the size is returned as 0. Image 0 may not return any information.

Return hardware revisions

The `ROM,F,V` command returns the firmware revisions of the Digital CPLD, Digital FPGA, Analogue CPLD and Analogue FPGA. The Micro1401 and Power2 and 3 have one CPLD and one FPGA and return 0 for `cpldA` and `fpgaA`.

Clear flash image

The `ROM,F,C,image` command clears a flash image. Factory set images are not cleared by this command.

Voltages and Waveforms

Capture and play-back of analogue signals is vital to many laboratory applications. The standard commands described in this chapter cover a wide range of tasks and should satisfy most users. We have developed many more commands, for higher performance but more specialised use; details are available on request.

Resolution

Old 1401s handle voltage signals to 12 bit accuracy, that is, 1 part in 4096. This is stored in the top 12 bits of 16 bit numbers with the lower 4 bits set to 0. Some applications do not need this resolution, and you can save space by choosing to use only the top 8 bits, giving a resolution of 1 in 256. ADC readings cannot resolve to better than half a bit, so conversion of a steady signal may fluctuate by the least significant recorded bit (even if there was no noise present). The Power1401 has 16-bit ADC and DACs and has gain options. The Micro1401 mk II has a 16-bit ADC and 12-bit DACs. The Micro1401-3 has 16-bit DACs.

Number of bits	Numeric range of data			Voltage range of data		
	low	high	Step	low	high	Step
16	-32768	32767	1	-5.000 V	4.9998 V	0.153 mV
12	-32768	32752	16	-5.000 V	4.9976 V	2.4 mV
8	-128	127	1	-5.000 V	4.961 V	39 mV

Impedances and voltages

You should consult the *Owners handbook* that came with your 1401 to get the full details of the inputs, pin numbers and electrical characteristics of your interface. The figures given here are for general guidance only.

The input impedance of the ADC inputs is typically not less than 1 MOhm and the standard full scale voltage range is ±5 Volts. The Micro1401 and Power1401 can be adjusted to ±10 Volts (see the *Owners handbook*).

A range of optional internal amplifiers is available for the Power1401, offering fixed or programmable gain, with or without filtering, to customise the 1401 to your particular voltage requirements. These do not affect the commands in this chapter in any way.

The full scale output voltage ranges are similarly ±5, or ±10 Volts and the amplifiers can drive 600 Ohms to 5 Volts, so they can drive headphones or 600 Ohm attenuators. They are not meant to be power drivers, however, and the settling time suffers with load impedances lower than 2 kOhms. The outputs are short circuit proof. Should you be unfortunate enough to have your inputs or outputs damaged, you will find that these devices are in sockets, for easy replacement.

The Micro1401 analogue inputs 0-3 are fed by front panel BNC connections. Expansion units are available to extend the number of inputs. The Power1401 has 8 analogue inputs and 2 DAC outputs on the front and 8 more inputs and 2 more outputs on the rear.

Power1401 ADC channels 8-15, DACs 2-3

Channel	8	9	10	11	12	13	14	15	DAC2	DAC3	Ground
Pin number	28	29	30	31	32	33	34	35	36	37	1-19

Selection of command

There are many 1401 commands that measure voltages and waveforms. To take a single measurement of the voltages on one or more channels, see the *Simple voltages* section. If you need to take a sequence of values on each channel, see the *Waveforms* section.

Simple voltages

To measure a set of up to 32 voltage channels, the easiest way is to use the ADC command. Sending this command, which is built-in and does not need to be loaded from disk, makes the 1401 measure the voltage on each of your list of channels, putting the results in the 1401 output buffer for the host to read when it has time.

You can set output voltages in a similarly way with the built-in DAC command.

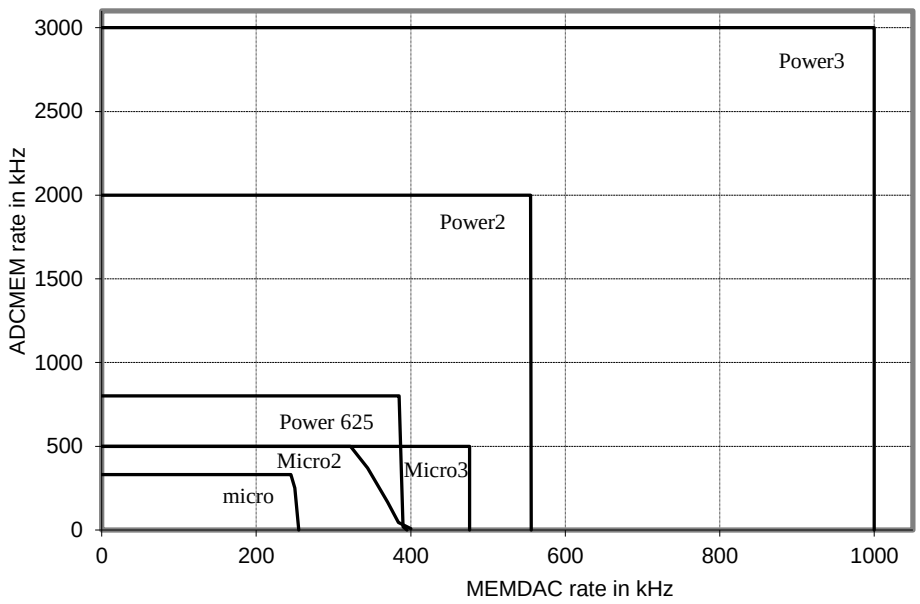
Waveforms

The ADC command is not suitable for sampling a waveform because the time between readings is not constant and it may be too slow, being limited by the rate at which the host can send the strings that fire it off, and the time it takes to read back the results.

There is a range of commands to read waveforms, each designed for a different pattern of use. To make your selection, you must decide first whether you need a multi-tasking command or conversely, whether you need the maximum speed, sacrificing the opportunity of multi-tasking.

This graph shows measured maximum multi-tasking rates using ADCMEM and MEMDAC commands for combinations of (single channel, 2 byte) ADC input and DAC output, for all current members of the 1401 family. The graph shows practical measurements and should be typical but it does not show guaranteed rates.

1401 family maximum ADC versus DAC rates



If you have more than one signal to measure, you must choose between even spacing of the channel samples in time, or reading them in a burst. Even spacing gives the fastest rates using the ADCMEM command, but if you need to know as closely as possible what value all the signals were when the clock ticked, you should choose the ADCBST, PERI32 or MADCM commands. PERI32 allows pre-trigger points to be sampled, MADCM can sample different channels at different rates.

The most commonly used command for reading a waveform is called ADCMEM. It takes a sequence of readings from your list of channels, evenly spaced in time, and puts the

results in an array in the 1401 memory. It can be used either in sequential form, or as a multi-tasking command - that is, running at the same time as other commands. You start it going, and later, tell it to stop.

There is a similar output command, called MEMDAC, which you can use to generate waveforms from data that you have stored in the 1401.

Note that as soon as a multi-tasking set up command is issued for the ADC (or DAC), that device is booked for use and any other call to use it will give an error.

Data rates

Rates in this table are for single channels (except ADCMEM), 2 byte data and are in kHz. The maximum rates may be diluted by other interrupt activities. ADC rates are aggregate rates for multiple channels unless otherwise stated. See the individual commands for detailed rates

	Micro1401			Power1401				
	mk I	mk II	-3	625	II,-3	-3	Comments	Page
ADC	~10	~10	~10	~10	~10	~10	Built-in command	26
ADCMEM	333	500	500	625	1000	1000	Multi-channel	28
				800	2000	3000	Single channel	
MADCM	125	500	500	384	1000	1000	Multi-channel multi-rate	31
ADCBST	333	500	500	625	1000	1000	Bursts of conversions	33
PERI32	222	500	500	625	1000	1000	Data-triggered bursts	36
DAC	~10	~10	~10	~10	~10	~10	Built-in command	27
MEMDAC	250	423	476	384	555	1000	Standard command	38

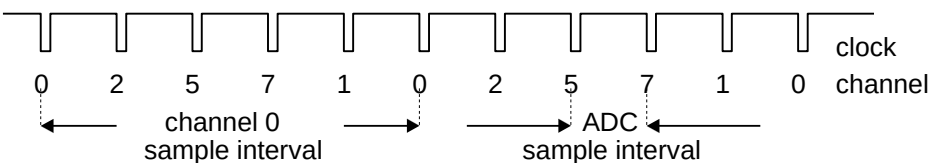
The 1401 does not check the rates you request, but speed failures are noted in hardware and shown by the automatic setting of a status flag, or error 253 or 31.

Some users need to capture more data than the 1401 can hold in its memory and will wish to transfer data to the host. The rates of transfer to the host depend on the interface card, the speed of the 1401 and the speed of the host.

There is a strong family resemblance between these commands, and the analogue output commands mirror the corresponding input commands, except for one important difference. In the input command ADCMEM, only one sample is taken at each tick of the clock, but in MEMDAC, all the specified DAC outputs are updated simultaneously at each tick.

The ADCBST and PERI32 commands read in bursts, with two channels taken exactly at the same time with the optional second sample and hold. This means the output may not be an exact replay of the capture, as the input readings were taken sequentially, not all at the same time. In the diagram we show 2 cycles round input channels 0, 2, 5, 7 and 1 during ADCMEM sampling.

Channel cycling during ADCMEM sampling



Clocks

The waveform input commands can be set to be driven directly by the ADC External Convert input, but usually a clock source will be chosen, with a divide down factor, and an optional triggered start. The setting of the clock is done within the analogue I/O command. Clock 3 is used for the output (DAC) commands, clock 4 for the ADC.

The clock source may be derived from the internal crystal, or an external clock signal of up to 10 MHz on the rear panel F input. The conversions occur at a rate set by this source, divided by the programmable scale factor.

The start of waveform input may be delayed, for synchronisation with a low going pulse on the rear panel E3 or E4 input. You can also route the front panel Trigger input to these inputs, see the `EVENT` command. After the first such pulse, the clock is no longer affected by this input. We call this a Triggered start.

For both the F (frequency) and E (event) inputs, the signal should be TTL. If the use of low going pulses on the E input is inconvenient, the active direction of the pulse can be inverted using switches, as described in the *Switch settings* chapter of the *Owners handbook*, or by the built-in `EVENT` command.

Standard arguments

The standard arguments in the various waveform input and output commands are described below. In the sections on the commands, only differences from these descriptions, and special arguments, are given.

`kind` Selects the mode: `I` for interrupt-driven, `F` for sequential*. If a sequential command is run in triggered mode, and a trigger is never given, the 1401 will hang up until a hardware reset! Sampling is always interrupt-driven, even in sequential mode, so the rates are the same. We also allow `IN` and `FN` for `kind` in `ADCMEM` and `ADCBST` to force a single sample to be taken per interrupt (useful in external convert modes when slow rates are expected).

`byte` Set either to 1 for 8-bit data (deprecated), or 2 for 16-bit data. The 8-bit data mode is only present for backwards compatibility with obsolete 1401s. It works, but is less efficient than 16-bit mode as no effort is made to optimise it.

`st` The start point for the data array in user space in the 1401. This is the byte address of the start and must be a multiple of the `byte` argument (1 or 2).

`sz` The number of bytes in the array. For 8 bit data, this is the number of data points. For 16 bit data this is twice the number of data points. For interrupt-driven commands the array is split into two halves internally so the number of data points must be even, i.e. `sz` must be `2*n*byte` where `n` is an integer.

For multi-channel use, `sz` must be divisible by (number of channels) * `byte`. If `sz` is not a sensible size, error 253 is given and the command will not run.

`chan` The list of channels in multi-channel commands. The maximum number of channels allowed depends on the command and the 1401 type. The channel numbers are separated by spaces. Channel -1 can be used for the DACs to skip over unwanted channels in arrays. All DACs are updated simultaneously, contrast the ADC inputs. We also allow a list holding a single channel in the range -121 to -128 to read special test channels.

Commands that support multiple ADC channels allow `-N` (`N` = 1 to 31) to mean a list of channels from 0 to `N` inclusive.

`rpt` The number of times to cycle round the array, range 1 to 4,294,967,295 or 0 for the maximum number of repeats (4,294,967,296).

`clock` The clock source is selected by a single letter:

- `T` for the internal 10 MHz source (`T` = Ten), (Micro1401 and Power1401 only)
- `H` for the internal 4 MHz source (`H` = High speed)
- `C` for the internal 1 MHz source (`C` = Clock)
- `F` for whatever TTL compatible signal is on the F rear panel input for DAC or ADC (F=Frequency). The F input for the Micro1401 for the ADC clock is ADC Ext, not F.
- `X` for replacing the clock by direct timing on the ADC Ext input or F rear panel input.

Add `T` (e.g. `CT`, `HT`, `TT` or `FT`) to suspend operation until the relevant E3 (DAC) or E4 (ADC) input is pulsed low. See command descriptions for the use of `XT`.

Some commands allow `R` (repeated trigger) in place of `T`, for example `ADCMEM`.

`pre` `pre*cnt` sets the clock divide down from the selected source. The range of values for both is 2 to 65535 works for all 1401 types. Power1 allows a `pre` value of 1 and Micro2/3 and Power2/3 allow both `pre` and `cnt` to be 1. Other values cause error 253. Example: to make the clock tick at 100 Hz using the 1 MHz source a divide down of 10,000 is required. This could be achieved with a `pre` of 2 and a `count` of 5,000.

`cnt` See the `pre` description.

* The standard 1401 had two modes, `I`nterrupt and `F`ast. Fast mode captured data in a dedicated loop as fast as possible, preventing multi-tasking. Later 1401s do all sampling in multi-tasking mode and emulate `F` by waiting for sampling to end.

ADC
Read a list of voltages

The built-in `ADC` command tells the 1401 to read the requested group of ADC channels and send the results to the host. The channel list can contain up to 32 channels, however the result of converting channels that do not exist is undefined. The general form is:

```
ADC,chan[,byte];values
```

If `byte` is omitted, it defaults to 2. For example, to read ADC channels 0 and 2 into two variables:

Example 3: Immediate reading of ADC

Open communication with 1401
Send this string:
```
ADC,0 2
```
Read back two numbers into variables
Close communication with 1401

GAIN
Control ADC gain

Power1401 and Micro1401 users can control the ADC gain (if the gain option is fitted) with the `GAIN` command. The command does not exist in other 1401 types.

`GAIN,N,chan;num`	Number of gain settings for `chan`
`GAIN,W,chan,gIndex`	set new gain index, ignored for channels with no gain
`GAIN,R,chan;gIndex`	read back gain index of a particular channel
`GAIN,S,chan,gain`	Set channel gains as near as possible to `gain` as % (100=unity)
`GAIN,G,chan;gain`	read back gain of a particular channel as % (100=unity)
`GAIN,L,chan;g0,g1,g2...`	read gain list for `chan` as % (100=unity)
`GAIN,B,n;nGain,chSt,chEnd,g0,g1,g2...`	read back gain block n information
`GAIN,M,chan;mV`	return the maximum ADC input for full scale at a gain of 1

Set channel gain (index)

`GAIN,W,chan,gIndex` sets the gain for channels in the channel list set by `chan`. The gains are set by an index; 0 is the lowest index, which usually corresponds to a gain of 1. You can get a list of available gains with the `L` or `G` options. `GAIN,S,chan,gain` sets the gain as near to `gain` as possible. A gain of 1 is 100, 3 is 300 and so on.

Read channel gain (index)

`GAIN,R,chan;gIndex` returns the gain index for a particular channel. The lowest index is 0. If a channel has no gain option, 0 is always returned. `GAIN,G,chan;gain` returns the channel gain as a percentage (100 = 1.0, 150 = 1.5 and so on).

Read number of gains

`GAIN,N,chan;num` returns the number of valid gain settings for a particular channel. It returns 0 if no gain option is fitted for this channel.

Get gain list

`GAIN,L,chan;g0,g1,g2...` returns the list of gains available for a particular channel. The gains are returned as a percentage: 100 is a gain of 1, 50 is a gain of 0.5 and 50000 is a gain of 500. You can get the number of gains from the `N` option. If no gain option is fitted, the return value is 100.

Read gain information

`GAIN,B,n;nGain,chSt,chEnd,g0,g1,g2...` returns information about a block of channels. Gain information is stored in the 1401 and in expansion units. Blocks of consecutive channels with the same gain options are grouped together. No information is returned for channels with no gain options. `n` is the block number to get information from, the first block is 0. `nGain` is the number of gain settings for the block, or 0 if the block does not exist. If this is returned as 0, all other values are 0 as well and no gains are returned. `chSt` and `chEnd` are the first and last channel in the block. The gains are returned as percentages in the same way as for the `L` option.

Get ADC range

`GAIN,M,chan;mV` returns the ADC range for the channel in mV (with the channel gain set to 1). Most units are set to ±5 Volt inputs, so the result is 5000. In a 10 Volt unit, the result is 10000. Normally, both ADC and DAC are set to the same range.

DAC
Set analogue output voltages

The built-in `DAC` command updates up to 4 output voltages. Micro1401s have 2 DAC channels only; writing to channels 2 and 3 has no effect but is not flagged as an error. If one channel is written to several times in the same command, pulses of 2.5 µs or shorter can be achieved. The general form is:

```
DAC,chan,values[,byte]
```

If `byte` is omitted, it defaults to 2. The following example sends the same voltage ramp (variable `x`) to all DACs 0 - 3. Micro1401 users will see output changes on DACs 0 & 1.

Example 4: Set voltage output levels

```
Open communication with 1401
FOR x = -128 TO 127                                  start a loop
  DAC,0 1 2 3,x x x x                    Value of x sent, see below
End of the loop
Close communication with 1401
```

Where a symbol (such as `x`) is used in an example and is shown as part of a string sent to the 1401, it is understood to stand for the value of the symbol as a number, the symbol itself is not sent. The first two strings sent to the 1401 in Example 4 are:

```
DAC,0 1 2 3,-128 -128 -128 -128
DAC,0 1 2 3,-127 -127 -127 -127
```

DGAIN
Control DAC gain

Power1401 and Micro1401 users can control the DAC gain (if the gain option is fitted) with the `DGAIN` command. The command does not exist in other 1401 types.

```
DGAIN,N,chan;num                    Number of gain settings for chan
DGAIN,W,chans,gIndex          set new gain index, ignored for channels with no gain
DGAIN,R,chan;gIndex                 read back gain index of a particular channel
DGAIN,S,chan,gain    Set channel gains as near as possible to gain as % (100=unity)
DGAIN,G,chan;gain          read back gain of a particular channel as % (100=unity)
DGAIN,L,chan;g0,g1,g2...          read gain list for chan as % (100=unity)
DGAIN,M,chan;mV          return the maximum ADC input for full scale at a gain of 1
```

Set channel gain (index)

`DGAIN,W,chan,gIndex` sets the gain for channels in the channel list set by `chan`. The gains are set by an index; 0 is the lowest index, which usually corresponds to a gain of 1. You can get a list of available gains with the `L` or `G` options. `DGAIN,S,chan,gain` sets the gain as near to `gain` as possible. A gain of 1 is 100, 3 is 300 and so on.

Read channel gain (index)

`DGAIN,R,chan;gIndex` returns the gain index for a particular channel. The lowest index is 0. If a channel has no gain option, 0 is always returned. `DGAIN,G,chan;gain` returns the channel gain as a percentage (100 = 1.0, 150 = 1.5 and so on).

Read number of gains

`DGAIN,N,chan;num` returns the number of valid gain settings for a particular channel. It returns 0 if no gain option is fitted for this channel.

Get gain list

`DGAIN,L,chan;g0,g1,g2...` returns the list of gains available for a particular channel. The gains are returned as a percentage: 100 is a gain of 1, 50 is a gain of 0.5 and 50000 is a gain of 500. You can get the number of gains from the `N` option. If no gain option is fitted, the return value is 100.

Get DAC range

`DGAIN,M,chan;mV` returns the DAC range for the channel in mV (with the channel gain set to 1). Most units are set to ±5 Volt inputs, so the result is 5000. In a 10 Volt unit, the result is 10000. Normally, both ADC and DAC are set to the same range.

The remaining commands in this chapter are not built-in. Unless they are loaded automatically on start up, you must make sure that you have loaded them yourself.

ADCMEM Equally spaced waveform input

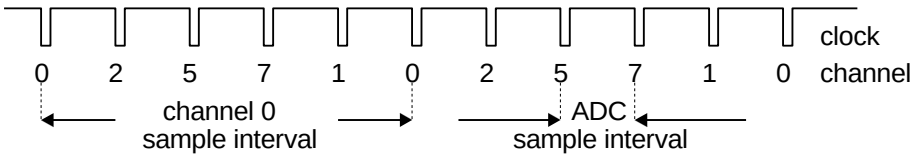
The ADCMEM command samples up to 32 ADC channels in either 8 or 16 bit format to an array in 1401 memory. It can run under interrupt control as a multi-tasking command, or in sequential mode. The maximum multi-tasking rate drops as other activities are increased; the graph on page 22 show the maximum measured rates for concurrent single channel 2 byte analogue input and output. This table shows the maximum multi-channel and single-channel rates in kHz with no other activity.

Maximum ADCMEM rates in kHz

	micro 1401	Micro 1401	Power Mk I	625	II,-3	-3
Interrupt	333	500	400	625	1000	1000
Single chan	333	500	2500	800	2000	3000

The command samples one ADC channel each clock tick. When there is more than a single channel to sample, the command cycles round the list of channels. If there are `n` channels and the clock is ticking every `t` µs, the interval between each sample on a single channel is `n * t` µs. If you require truly synchronous sampling on all channels you should consider the ADCBST command described on page 33. The diagram below shows the sequence when channels 0, 2, 5, 7 and 1 are selected.

Sampling times of inputs 0, 2, 5, 7 and 1.



When multiple channels are sampled, data is stored interleaved in memory, in the same sequence as the channels were sampled. See the SN1 and SN2 commands on page 59 for quick ways of extracting single channels. You are allowed to repeat a channel in a list, for example the channel list 0 1 0 2 0 3 could be used to sample channel 0 at three times the rate of channels 1, 2 and 3.

Command variants

```
ADCMEM,kind,byte,st,sz,chan,rpt,clock,pre,cnt      Clock set up
ADCMEM,kind,byte,st,sz,chan,rpt,X[T]               External convert set up
ADCMEM,K                                           Kill interrupt-driven sampling
ADCMEM,S                                           Stop interrupt-driven sampling at buffer end
ADCMEM,?;status                                    Report state of interrupt-driven sampling
ADCMEM,P;offs                                      Report next position to be updated
ADCMEM,Z;device,fnclvl,nch,adc                     Report channel sequencer configuration
```

`kind` is either of `I` for multi-tasking mode or `F` for sequential mode. The `rpt` argument must be present in both `ADCMEM,I` and `ADCMEM,F` forms but is ignored in the latter.

Clock set up

A command of this form will reset the ADCMEM command, and either start the sampling immediately, if clock is set to `T`, `H`, `C` or `F`, or wait for a low going TTL pulse on the clock 4 event input (E4) if `TT`, `HT`, `CT` or `FT` are used.

External convert set up

A command of this form will cause an ADC sample to be taken for each low going pulse received on the external convert input ADC Ext of the 1401. With the optional `T`, in the dedicated `F` version, data collection waits for an E4 start pulse. Care should be taken with the `T` option in dedicated mode; if no trigger pulse is received, the machine will freeze. This can be cleared by a hardware reset using the `Reset1401` Language Support routine.

Stop sampling

`ADCMEM,K` stops interrupt-driven sampling immediately, resets the ADC, and releases it for use by other commands. The sequential form cannot be stopped before `sz` bytes are taken because once `ADCMEM,F` is started, no other command, including `ADCMEM,K` can be run. To escape this impasse use the Language support function `Reset1401`.

`ADCMEM,S` stops the command when the buffer currently being processed is finished.

Query status

ADCMEM,? causes the sampling buffer status to be returned (for interrupt mode only) as:

-128	Sampling has not yet filled a half buffer
-1	Sampling finished, but some samples were missed (sampling was too fast)
0	Sampling has completed the set number of repeats correctly
1	Second half of buffer is filling, first half is free
2	First half of buffer is filling, second half is free

Report update position

ADCMEM,P returns the byte address (relative to st) of the next byte to be updated, either during logging, or after the command stops.

Return channel sequencer information

ADCMEM,Z;device,fnclvl,nch,adc is used to return information about the channel sequencer, for programs for general distribution that have to optimise performance. See the ADCBST description on page 33 for full details.

Error reports

Code	Reason
253,1	Bad number of samples per channel; not divisible by 2
253,2	Bad number of samples; sz/(chan*byte) is not an integer
253,3	Sampling is being driven too fast in dedicated mode

Example of triggered data capture

Sample four channels of data in dedicated mode, using 1 byte per point, into an array at 20 kHz (5.0 kHz per channel). Input is triggered by the E4 input. When complete, the data is returned to the host.

Example 5: Dedicated data capture

Put a message up on the screen to say we are starting, set up communication with the 1401 and load the ADCMEM command, if needed.
Clear any previous activity and set up triggered sampling of 1 byte data to an array from address 0, on channels 0 2 7 and 1 in that order, at 1 MHz divided by 5 * 10 by sending:

```
CLEAR
ADCMEM,F,1,0,2048,0 2 7 1,1,CT,5,10
```

Set a data transfer to xxxx in host memory when ADCMEM is complete by sending:

```
TOHOST,0,2048,xxxx
```

Check for completion by asking to read a variable (the error status) by sending the string:

```
ERR
```

Print a message to show we have finished.

Repeated trigger

ADCMEM repeated trigger mode collects multiple sweeps of data with the minimum time delay between the sweeps. There is no sampling speed penalty for the use of this mode.

Command variants

```
ADCMEM,kind,byte,st,sz,chan,rpt,clockR,pre,cnt[,swpsz]    Clocked
ADCMEM,kind,byte,st,sz,chan,rpt,XR[,swpsz]                External convert
```

These calls are identical to the standard setup calls, except that the clock control contains R (think Repeated trigger) in the trigger field and there is an optional additional field:

swpsz The byte size of each sweep of data to be captured. If omitted it takes the value sz/2. It may not be larger than sz, but it does not have to divide exactly into sz.

The data area defined by st and sz is divided into sub-buffers, each of size swpsz bytes. If sz does not divide exactly by swpsz, sz is rounded down to an exact multiple. Each sub-buffer is sampled in triggered mode (using the E4 trigger input to start sampling).

Multi-tasking mode notes

In `ADCMEM,I...` mode, the `rpt` field sets the sub-buffers to capture; sampling cycles round the entire buffer several times if this is necessary to complete the number of sub-buffers requested. The largest number of sweeps possible is collected by setting `rpt` to 0. This gives 2^{32} sweeps. If each sweep lasted only 1 millisecond, it would take nearly 50 days to complete the sampling task, so you can think of `rpt` = 0 as running for ever.

The `ADCMEM,P` command variant operates unchanged, returning the next position to be written to. The `ADCMEM,?` command returns the number of sub-buffers which have been sampled. It does not return 0 when sampling is over. The `ADCMEM,S` command will stop the sampling after the next sub-buffer has been captured. `ADCMEM,K` kills sampling immediately.

You would run in this mode when it is necessary to monitor the progress of the command, or when you need to sample more data than can fit into the 1401 memory. You can transfer data from the 1401 back to the host while the command is sampling, using the `ADCMEM,?` and `ADCMEM,P` commands to monitor the command progress.

Sequential sampling notes

In sequential sampling (`ADCMEM,F...`), the `rpt` field is ignored, and the entire buffer is filled once only with `sz/swpsz` sub-buffers. In this mode the only way to get control back from the 1401 before all the sub-buffers have been sampled is to send a hardware reset to the 1401 (see the `Reset1401()` routine in the appropriate *Language Support Manual*). In this case, the `ADCMEM,?` command will return the number of completed buffers before the reset.

Power1401 prior to Power1401 625

The ADC in the original Power1401 produces new data values at a constant rate of either 2.5 MHz or 10 MHz. This is different from the ADCs in the other 1401s (including the later Power1401 625) that produce new values on demand unless the demand exceeds the maximum rate. The important differences are in externally clocked mode, and when sampling single channels at rates above 400 kHz; otherwise the ADC behaves just the same as in all other 1401s.

When used in multiple channel mode, or single channel mode up to 400 kHz, we use the 10 MHz output rate and digitally filter the data in the ADC hardware to reduce high frequency (in the MHz range) noise. In triggered start mode, there will be up to 0.1 microseconds delay between the trigger and the first point. All subsequent points will be at the requested clock spacing. If you run in an externally clocked mode, you will get the first available sample after the external clock tick, that is all samples will be taken at the next 0.1 microsecond interval. Put another way, in externally clocked mode, there will be a jitter of ±50 nanoseconds on each sample. There is no jitter when the ADC is clocked internally.

When used in single channel mode at rates above 400 kHz the ADC is run in a different mode that produces outputs every 0.4 microseconds. In triggered start mode, there will be up to 0.4 microseconds delay between the trigger and the first point. If you request a clock rate that is a multiple of 0.4 microseconds (0.4, 0.8, 1.2, 1.6, 2.0 or 2.4 microseconds), you will get output sampled at precisely the requested rate. If you request output at any other rate between 0.4 and 2.5 microseconds, each point will take the next available ADC sample.

The ADC has very low noise at total sampling rates up to 200 kHz. The noise increases as the sampling rate approaches 400 kHz in the multi-channel case as the amount of high frequency digital filtering that can be done is reduced by channel switching times and the increasing sample rate. The noise is the same for all single channel sampling rates above 400 kHz.

MADCM Multi-rate waveform input

The multi-tasking MADCM command samples up to 32 ADC channels into separate buffers in 1401 memory at rates that can be different but that are all sub-multiples of a base rate. You can start and stop channels at will, and monitor individual channel status.

The sample times of all channels is normally clock controlled by the channel sequencer and in all 1401s except the standard, you will get faster maximum rates with the sequencer (unless you only sample a small fraction of the total channels). If you do not, use the sequencer you will get un-clocked sampling with some sample time jitter.

Channels 0 and 7 (0 and 3 for the micro1401) are sampled simultaneously when they are both called in a burst, if the dual sample/hold option is fitted, and if the channel list starts with 0 7 (0 3). If the clock is set too fast for the channel list, error 16 (31 for Power1401 and micro1401) occurs after the command runs. The maximum rates in kHz for 2 byte data, with all channels at the same rate, are:

Maximum MADCM rates in kHz

Channels clocked / no clock	**1**	**2**	**4**	**8**	**16**
micro1401	125 / 64.5	90.9 / 50	58.8 / 29.4		
Micro1401 mk II	500 / 238	250 / 136	125 / 75		
Micro1401-3	500 / 256	250 / 151	125 / 83		
Power1401	384 / 250	200 / 143	100 / 78	50 / 40	25 / 20
Power1401 mk II	1667 / 300	500 / 232	250 / 139	125 / 76	62.5 / 40
Power1401-3	2000 / 625	500 / 333	250 / 200	125 / 100	62.5 / 50

Command variants:

`MADCM,Z[,chan]`	Initialise the command
`MADCM,clock,pre,cnt`	Clock set up
`MADCM,Q`	Clock stop
`MADCM,An,ch,st,sz,dvd,rpt`	Channel set up
`MADCM,S,ch,rpt`	Change channel repeats
`MADCM,?,ch;status`	Read channel status

Initialise

`MADCM,Z[,chan]` should be used first. If `chan` is present, it sets the list of channels to use. If omitted, un-clocked mode is set. The clock is stopped if running. Using a channel list is strongly recommended.

Clock set up

`MADCM,clock,pre,count` will reset the channel list memory, and either start the sampling immediately, if clock is set to `T`, `H`, `C` or `F`, or wait for a low going TTL pulse on the clock 4 event input (E4) if `TT`, `HT`, `CT` or `FT` are used. This is the basic clock rate for the command; individual channels will be set at sub-multiples, see below. Once started, the clock will run forever unless stopped!

Clock stop

`MADCM,Q` stops the clock from running, and thus causes sampling to stop. The ADC will be released for use by the ADC command and for cursors in the `D` command. This has immediate effect.

Channel set up

`MADCM,An,ch,st,sz,dvd,rpt` links an individual channel into the `MADCM` sampling table and should be used before the clock setup command. If `MADCM,Z` has been used to set a list, to get crystal controlled sampling, the channel must only be taken from that list. With no list, any of the 16 may be used.

Channels may be sampled as one byte or two byte data; setting `A1` as the second field establishes 1 byte mode, `A2` sets 2 byte mode. The third field sets the channel number to be used, in the range 0 - 15.

The next two fields set the start address in the 1401 user area and the size of this area in bytes. The sixth field sets the divide down from the basic clock rate to be used by this channel. This divide down may be in the range 1 to 256. To set a divide down of 256, use 0; any value greater than 255 will cause an error to be flagged.

The final field is the number of times to fill the buffer defined by `st` and `sz`; this will usually be set to 1! If other values are used you must be prepared to copy filled buffers elsewhere in 1401 or to the host if you wish to preserve old data.

Change channel repeat count

`MADCM,S,ch,rpt` allows the number of channel repeats to be dynamically changed, as long as the channel is active. If the channel is inactive, or has finished sampling, then no effect will be observed.

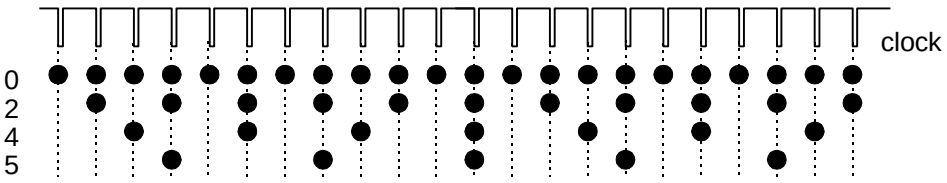
Read channel status

`MADCM,?,ch` causes the channel status to be returned as:

-128	Channel has not yet filled a half buffer
0	Channel has completed the set number of repeats
1	Second half of buffer is filling, first half is free
2	First half of buffer is filling, second half is free

Example of use of MADCM

This example sets up ADC channels 0, 2, 4 and 5 to be sampled at 100 Hz, 50 Hz, 33.3 Hz and 25 Hz. The channel 0 result is 1024 bytes long, channel 2 is 512 bytes long, channel 4 is 340 bytes and channel 5 is 256 bytes long. The sampling sequence is:



All the channels are one byte data. The basic clock rate is set to be 100 Hz, by setting a divide down of 10 * 1000 from the 1 MHz system clock, and the sampling is to start by a trigger on E4. The list of channels is not given, so sampling within a burst will not be strictly synchronous. With this choice of sampling rates and array lengths, sampling will stop on all channels at the same time, though this is not necessary in general use.

Example 7: Multi-rate input

Print a start message on the host screen, and set up communication with the 1401. Clear 1401, ignore the channel sequencer, stop the clock and set up channels 0 2 4 and 5 by sending these strings (note the staggered start addresses):

```
CLEAR
MADCM,Z
MADCM,Q
MADCM,A1,0,0,1024,1,1
MADCM,A1,2,1024,512,2,1
MADCM,A1,4,1536,340,3,1
MADCM,A1,5,1876,256,4,1
```

Now set the clock rate and start it by sending:

```
MADCM,CT,1000,10
```

Check for completion by sending this string and reading the value of the status of channel 0, until it becomes zero:

```
MADCM,?,0
```

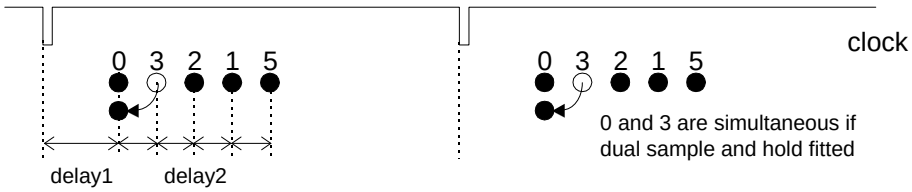
Return communication to the screen, if needed, and print a message to show it is all over.

ADCBST
Burst mode waveform sampling

The ADCBST command samples ADC data in burst mode. In this mode all of the ADC channels in the list are sampled in a crystal controlled burst, as close together as possible, in contrast with the ADCMEM commands, in which samples are evenly spaced in time.

The optional second sample and hold allows bursts of samples to be simultaneously sampled on channels 0 and 3 in the micro and Micro2, if they are first and second in the channel list. Operation reverts to normal if any other ADC sampling command is used.

This diagram shows the sampling times for channels 0, 3, 2, 1 and 5 with the optional second sample and hold if you use a micro1401 or Micro1401 mk II.



The burst of conversions is triggered by a clock 4 tick or an external signal. The command can be used in interrupt driven mode to allow concurrent use of other 1401 commands or in dedicated mode for maximum speed.

The ADCBST command is included in the ADCMEM command, and has all the features of ADCMEM, including repeated triggers. If your application uses both ADCMEM and ADCBST, load ADCMEM first. If it uses only ADCBST, you must load ADCMEM.

Command variants

`ADCBST,kind,byte,st,sz,chan,rpt,clock,pre,cnt`	Clocked start
`ADCBST,kind,byte,st,sz,chan,rpt,X`	External triggered start
`ADCBST,T,delay1[,delay2]`	Set burst timing
`ADCBST,?;status`	Read the command state
`ADCBST,P;offs`	Read buffer pointer
`ADCBST,S`	Stop sampling smoothly
`ADCBST,K`	Kill all sampling
`ADCBST,Z;device,fnclvl,nch,adc`	Report channel sequencer configuration

Arguments are used as described above in 'Standard arguments', except that `chan` has been extended:

`chan` List of channels to be sampled, in order. If the number of channels does not divide into the buffer size, error 253,127 is reported. Channels numbers up to the maximum fitted can be used. A negative number (`-N`) as the only entry means use channels 0 to `N`, where `N` can be up to the maximum channel number.

Set up and start sampling

`ADCBST,kind,byte,st,sz,chan,rpt,clock,pre,cnt` This command initiates the sampling at a rate set by the `clock`, `pre` and `count` parameters. Every `pre * cnt` clock source periods a burst of ADC samples will be taken, one for each channel in the channel list. The ADC data is stored in interleaved form in the memory array. Sampling will continue until `rpt` cycles round the array have been completed or until the command is stopped. Both the `ADCBST,I` and `,F` forms use interrupt driven sampling. The `,F` form waits for sampling to end before returning.

This table shows typical maximum rates per channel for 2 byte data, with no other commands running. These speeds assume that the best values of `delay1` and `delay2` have been selected.

Maximum ADCBST sample rates

channels	1	2	4	8	16	32
micro1401	333	167	83			
Micro2/3	500	250	125			
Power1401 mk I	2500	200	100	50	25	12.5
Power1401 625	800	313	156	78	39	19.5
Power1401 II	2000	500	250	125	62.5	31.25
Power1401-3	3000	500	250	125	62.5	31.25

Set up externally converted sampling

`ADCBST,kind,byte,st,sz,chan,rpt,X` samples on the ADC external convert input. Every pulse on the input initiates a burst of samples. In all other ways this form of the command is identical to the clocked conversion form.

Set the timing parameters

`ADCBST,T,delay1[,delay2]` sets the timing parameters for the burst in channel sequencer clock periods. The hardware sequencer in the micro1401 runs with a 0.25 µs clock period. The Micro1401 and Power1401 run at 0.05 µs.

`delay1` sets the delay between clock 4 ticks or external pulse and the first conversion. Any value up to 255 can be used. The minimum is 3 for the Power2 or Power3 and 5 for the micro and Power1. `delay1` is normally set to the minimum value to give the fastest possible throughput.

`delay2` sets the interval between successive conversions in the burst. Any value up to 255 can be used. You should not set this to be less than your ADC conversion time (3 µs for the micro1401 and 2.5/1.6/1.0 µs for the Power1401 mk I/ 625/Mk II). *Power1401 mk 1 only*: `delay2` values less than 56 cause problems on channel n+1 if channel `n` exceeds the 1401 input range.

If `delay2` is not provided, only `delay1` is set. Any alterations made are used by the command until they are altered again by another use of the `ADCBST,T` command or until the command is reloaded into the 1401. If values shorter than the minimum values are used, the clocking will not be reliable.

For the minimum delays and fastest burst rates, set the values in the table to the right. As all 1401s automatically select the best values by default, you only need set values if you have changed them away from the defaults.

1401	delay1	delay2
micro	5	12
Micro2/3	3	40
Power1	5	50
Power2/3	3	20

Return channel sequencer information

`ADCBST,Z;device,fnclvl,nch,adc` returns channel sequencer information, for use in programs that have to optimise performance.

`device` This returns a code for the channel sequencer fitted. A micro1401 has code 2 and the Power1401 and Micro1401 have code 3.

code	type	µs
2	Hardware	0.25
3	Hardware	0.05

`fnclvl` This is the functionality of the sequencer firmware (a minimum of 2).

`nch` The number of channels that the sequencer can accept. This is a minimum of 32. Selecting channels above those fitted to your 1401 results in a command error.

`adc` This is a code for the ADC block. The table lists accuracy and speed in kHz for single and multi-channel use for current ADC devices.

adc	single	multi	bits	Used in 1401s
4	2500	400	16	Power1401 mk I
5	500	500	16	Micro1401
6	800	625	16	Power1401 625
7	3000	1000	16	Power1401 mk II/-3

Find the status of the sampling

`ADCBST,?;status` is used to return information in a similar format to the other high performance waveform input commands:

`-128`	The first half buffer is not yet filled
`-1`	Sampling finished, but some samples were missed
`0`	The set number of repeats are completed correctly
`1`	The second half buffer is filling, the first is free
`2`	The first half buffer is filling, the second is free

Where is the data currently going?

`ADCBST,P;offs` returns `offs` as the current value of the buffer pointer, as in `ADCMEM,I`. The value is the byte offset from the start of the specified data area to the next location to be written to.

Stop sampling

`ADCBST,S` stops the sampling smoothly the next time the end of the buffer is reached. `ADCBST,K` terminates all sampling immediately.

PERI32 peri-event triggered waveform sampling

PERI32 captures up to 32 waveform channels. Data is sampled under interrupt into a buffer until a trigger condition is detected. A set number of additional post-trigger data points are captured and sampling stops. Data is retrieved by a PERI32 command variant that calculates the position of data in the circular buffer area.

PERI32 samples data in burst mode, with simultaneous sampling of channels 0 and 3 with micro1401 and Micro2 if these are the first in the list and the simultaneous sampling option is fitted.

Maximum PERI32 data rates in kHz (no other activities)

Channels	1	2	3	4	n
micro1401	222	166	111	82	330/n
Micro1401	500	250	167	125	500/n
Power1401	769	200	133	100	400/n
Power1401 625	1000	313	208	156	625/n
Power1401 mk II	1666/909	500	333	250	1000/n
Power1401-3	2000	500	333	250	1000/n

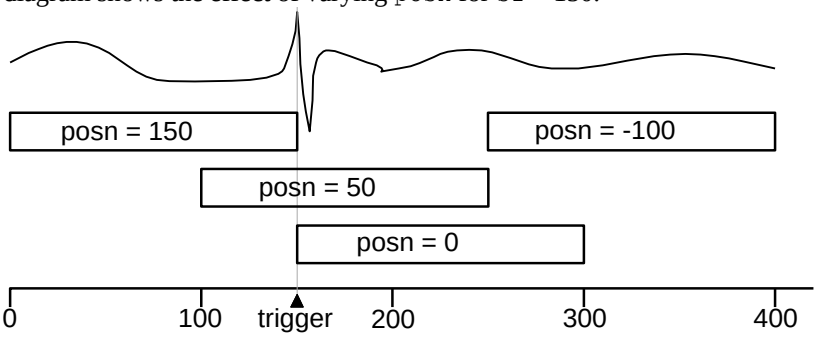
The trigger for sampling can be a pulse on the event 0 input, an ADC signal passing a threshold, or a TTL level on a digital input. Where 2 rates are given, the slower rate is with a digital trigger on bits 0-7; all other triggers can achieve the higher rate.

Command variants

`PERI32,byte,st,sz,chan,posn,clock,pre,cnt`	Set up sampling
`PERI32,E`	Set event trigger
`PERI32,A+,level or PERI32,A-,level`	Set analogue trigger
`PERI32,D0,bit or PERI32,D1,bit`	Set digital trigger
`PERI32,G`	Start sampling
`PERI32,P;amount`	Return bytes available
`PERI32,H`	Hang until finished
`PERI32,?;state`	Query sampling state
`PERI32,K`	Kill sampling
`PERI32,F,hoOff,st,sz`	Fetch sampled data
`PERI32,T,delay1[,delay2]`	Set sequencer timing
`PERI32,Z;device,fnclvl,nch,adc`	Get sequencer information

Non-standard arguments

`posn` The position in the result array of the ADC data collected at the trigger point. It may range from any negative value to a positive value less than `sz`. It is a byte address and must be an exact multiple of the number of channels times `byte`. The diagram shows the effect of varying `posn` for `sz` = 150.



`level` The trigger voltage level, set as a 2 byte ADC value from -32768 to 32767.

`bit` A value specifying a digital input bit, from 0 to 15.

Setting up data handling

`PERI32,byte,st,sz,chan,posn,clock,pre,cnt` stops any ADC use and sets the sampling parameters, the memory area, the trigger position and the clock source.

Setting the trigger mode

The trigger mode must be selected before sampling starts. Trigger options are:

`PERI32,E` the trigger will be the event 0 input. This is normally a low-going transition but the 1401 can be set up to respond to high-going transitions instead, see `EVENT`.

`PERI32,A+,level` the trigger is a low-to-high threshold crossing of the last waveform channel in the `chan` parameter list in the `PERI32` setup. The `level` parameter sets the threshold, in ADC units from -32768 to 32767, for both 8 and 16 bit data acquisition.

`PERI32,A-,level` is the same as `PERI32,A+,level` with a high-to-low transition.

`PERI32,D0,bit` the trigger is a digital input bit reading as zero. `bit` defines the bit to be used, from 0 to 15. The digital trigger must be in the valid state for longer than the time between bursts, set by `pre` and `count`.

`PERI32,D1,bit` is the same as `PERI32,D0,bit` except that the bit must read as 1.

Run time control

`PERI32,G` starts sampling, using the parameters defined in the two setup commands. It may be used more than once with one set of input parameters, avoiding the need for multiple set up calls. Once started, sampling continues until halted by a trigger or by:

`PERI32,K` stops sampling, and resets the ADC and the ADC clock. The data stored in memory is not usable after this command. It does not alter setup parameters stored in memory, so it is possible to run `PERI32,G` after `PERI32,K`.

`PERI32,H` hangs all activity in 1401 until the `PERI32,G` sweep is done, like `ADCMEM,F`.

`PERI32,?;status` returns a value to indicate the current status of the command:

2	Sampling primed and waiting for trigger
1	Trigger detected, sampling in progress, waiting for sweep end
0	Sweep finished, or not started yet
-128	Sampling not started yet, waiting for an event 4 pulse

Attempts to drive the sampling too fast give error 31.

Report sampled bytes

`PERI32,P;amount` returns the number of bytes of ADC data available for transfer to the host with the `F` sub-command. If `amount` is less than `sz`, data transferred with the `F` sub-command is interim data only, and may be different if you ask for the same data again. Once the command has triggered, any data you fetch with the `F` sub-command will remain unchanged, even after the command has finished sampling.

Collect sampled data

`PERI32,F,hoOff,st,sz` transfers data to the host. The `st` parameter is the offset into the data to start transferring from, and `sz` is the number of bytes to transfer. `st` is not a memory address but an offset relative to the first data point available. The value of `st + sz` should not exceed the `amount` value returned by the `P` sub-command or the `sz` parameter used in the setup. For example, if the command is sampling into an area 4096 bytes in size, an `F` command with `st` and `sz` of 0 and 1024 will transfer the first quarter of the data. The data is transferred to the host memory area identified by `hoOff` (see page 17 for more discussion of `hoOff`). Requesting too much data causes error 253,64.

Channel sequencer control

`PERI32,T,delay1[,delay2]` and `PERI32,Z;device,fnclvl,nch,adc` control the ADC channel sequencer. These commands are identical to the `T` and `Z` subcommands of `ADCBST` to which you should refer for a full description.

MEMDAC waveform output

The MEMDAC command plays 1401 memory to any combination of the DAC channels, at a user-defined rate. If more than one channel is required, the data must be interleaved in memory to appear on the correct output. All the channels are output on each clock tick, and are updated together. MEMDAC uses clock 3.

MEMDAC maximum rates kHz

Channels	1	2	3	4	8
micro1401	250	250			
Micro1401 mk II	500	455			
Micro1401-3	500	476			
Power1401	384	357	344	344	
Power1401 mk II	555	500	455	416	313
Power1401-3	2000	2000	2000	2000	1000

The interrupt rates assume no competing interrupts. MEMDAC does not limit the rate you ask for, but over-ambitious rates cause error 31.

Command variants

`MEMDAC,kind,byte,st,sz,chan,rpts,clock,pre,cnt`	Clock set up
`MEMDAC,kind,byte,st,sz,chan,rpts,X[T]`	External [triggered] set up
`MEMDAC,S`	Stop after current sweep
`MEMDAC,K`	Kill current play
`MEMDAC,?;status`	Query command state
`MEMDAC,P;offs`	Report next byte to be played

Clock set up

Once the clock setup command has been issued, the 1401 will either begin to play the area of memory immediately, or if the triggered option has been selected, it will wait for a low going pulse on the E3 input. The mode is set by `kind`.

Dummy channels can be put in the `chan` list, with channel numbers of -1. This enables replaying channels from a record of a larger number of channels.

External triggered set up

This is identical with the variant above, except that the DACs are updated directly on pulses received on the rear panel F input.

Stop playing

`MEMDAC,S` stops output after the next sweep. This will not cause any additional output if the play has already terminated. `MEMDAC,K` kills any active play, and resets the command software and hardware.

Query status

`MEMDAC,?` causes the command status (multi-tasking mode only) to be returned as:

-128 Command has not yet output the first half buffer
0 Command has completed the set number of repeats correctly
1 Second half of buffer is playing, first half is free
2 First half of buffer is playing, second half is free

Report position

`MEMDAC,P` returns the position of the next byte for output. The DACs are used in double-buffered mode; the pointer shows the next byte to be taken from memory, not be the next value to be converted. Double-buffered means that values written to the DAC outputs do not cause the outputs to change until the next clock tick or external pulse. This allows us to update all the DACs together (synchronously) even though we have to write values to them at different times. When using the DAC Silo with recent 1401s, the position can be many samples ahead of the current values on the DAC outputs.

Example of use

This example will play 1000 sweeps of the contents of the first 1024 bytes of memory, treated as 2 byte data at 10 kHz, being 1 MHz divided by 10 * 10.

Example 8: MEMDAC output in interrupt mode

```
Print a message on the host screen to say we are starting
Open1401                                  Open communication with the 1401
Load MEMDAC command, if not already loaded
        CLEAR                                    Stop any 1401 activity
        MEMDAC,I,2,0,1024,0,1000,C,10,10         Start output
        MEMDAC,?                  Request status, wait for 0 meaning done
Close1401                                 Close communication with the 1401
```

Repeated trigger

MEMDAC repeated trigger mode plays multiple sweeps of data with the minimum time delay between the sweeps. There is no sampling speed penalty for the use of this mode.

Command variants

```
MEMDAC,kind,byte,st,sz,chan,rpt,clock,cnt[,swpsz]      Clocked
MEMDAC,kind,byte,st,sz,chan,rpt,XR[,swpsz]             External convert
```

These calls are identical to the standard setup calls, except that the clock control contains R (think Repeated trigger) in the trigger field and there is an optional additional field:

`swpsz` The byte size of each sweep of data to output. If omitted it takes the value `sz`/2. It may not be larger than `sz`, but it does not have to divide exactly into `sz`.

The data area defined by `st` and `sz` is divided into sub-buffers, each of size `swpsz` bytes. If `sz` does not divide exactly by `swpsz`, `sz` is rounded down to an exact multiple. Each sub-buffer is output in triggered mode (using the E3 trigger input to start output).

Multi-tasking mode notes

In this mode, the `rpt` field defines the number of sub-buffers to output; it cycles round the entire buffer several times if this is necessary to complete the number of sub-buffers requested. To collect the largest number of sweeps possible, set `rpt` to 0. This gives 2^{32} sweeps. If each sweep lasted only 1 millisecond, it would take nearly 50 days to complete the sampling task, so you can think of `rpt` = 0 as running for ever.

The `MEMDAC,P` command variant (interrupt mode only) operates unchanged, returning the next position to write to the DACs. The `MEMDAC,?` command returns the number of sub-buffers output. It does not return 0 when output ends. The `MEMDAC,S` command stops the output after the next sub-buffer. `MEMDAC,K` kills output immediately.

You would run in this mode when it is necessary to monitor the progress of the command, or when you need to play more data than can fit into the 1401 memory. You can transfer data to the 1401 from the host while the command runs, using the `MEMDAC,?` and `MEMDAC,P` commands to monitor the command progress.

Sequential mode notes

In this mode (`MEMDAC,F...`), the only way to get control back from the 1401 before all the sub-buffers have been sampled is to send a hardware reset to the 1401 (see the `Reset1401()` routine in the appropriate *Language Support Manual*). In this case, the `MEMDAC,?` command returns the number of completed buffers before the reset.

Switching, counting and timing

This chapter describes the digital input and output lines and their control, clocks 0, 1 and 2 and the event inputs.

The digital port

There are 32 digital I/O bits; 16 permanent outputs and 16 permanent inputs. The connectors are located on the rear panel.

Digital i/o connectors

Pin	Function (output socket)	Pin	Function (input plug)
1	Output 15	1	Input 15
14	Output 14	14	Input 14
2	Output 13	2	Input 13
15	Output 12	15	Input 12
3	Output 11	3	Input 11
16	Output 10	16	Input 10
4	Output 9	4	Input 9
17	Output 8	17	Input 8
18	Output 6	18	Input 7
5	Output 7	5	Input 6
6	Output 5	6	Input 5
19	Output 4	19	Input 4
7	Output 3	7	Input 3
20	Output 2	20	Input 2
8	Output 1	8	Input 1
21	Output 0	21	Input 0
9	Data received input DR	9	Data transmitted o/p (8-15) DT
22	User input (buffered, reserved)	22	- not connected -
10	User output (buffered, reserved)	10	- not connected -
23	New Data Ready (0-7) NDRL	23	Data Available i/p (0-7) DAL
11	Output enable bits 8-15	11	- not connected -
24	Data received i/p (0-7) DRL	24	Data transmitted o/p (0-7) DTL
12	New Data Ready (8-15) NDR	12	Data Available i/p (8-15) DA
25	+5V (250 mA maximum)	25	+5V (250 mA maximum)
13	GND	13	GND

Electrical specifications of the inputs and outputs for each member of the 1401 family can be found in the *Owners handbook* for each model.

DIG simple digital i/o

The built-in DIG command controls the digital port. It is intended for occasional changes; if you need to send out a regular time varying pattern on the digital outputs, the DIGTIM command, described later in this chapter is much better.

Command variants

```
DIG,I[,bit];value        Read the input
DIG,O,value[,bit]        Set the output
```

`value` is a number: byte or word as appropriate.

`bit` (if present) is the number of the bit in the 16 bit word.

Reading the digital inputs

`DIG,I[,bit];value` will read the digital input word and report either the state of an individual bit, or the entire word. Any bits in the low byte that are being used as output will read as 0. To read one bit, include the optional [,bit]. The entire word is read if the bit number is omitted. Thus:

`DIG,I;value` reads the entire 16-bit input word, result in the range 0 to 65535
`DIG,I,0;value` reads the state of bit 0 only

Setting the digital outputs

`DIG,O,value[,bit]` operates in word or bit mode according to the number of arguments sent. In word mode, it transfers the upper byte of `value` to outputs 8 to 15 and the lower byte to any of the bits 0 to 7 set to be outputs.

In bit mode, it transfers the state of bit 0 of `value` to the bit number specified by `bit`. For example:

`DIG,O,$FF34` sends the hex value FF34 to the output
`DIG,O,1,15` sets bit 15 of the output word

Example

This program will 'ramp' the digital outputs (send a steadily rising binary number) and stop if bit 15 of the digital input is set:

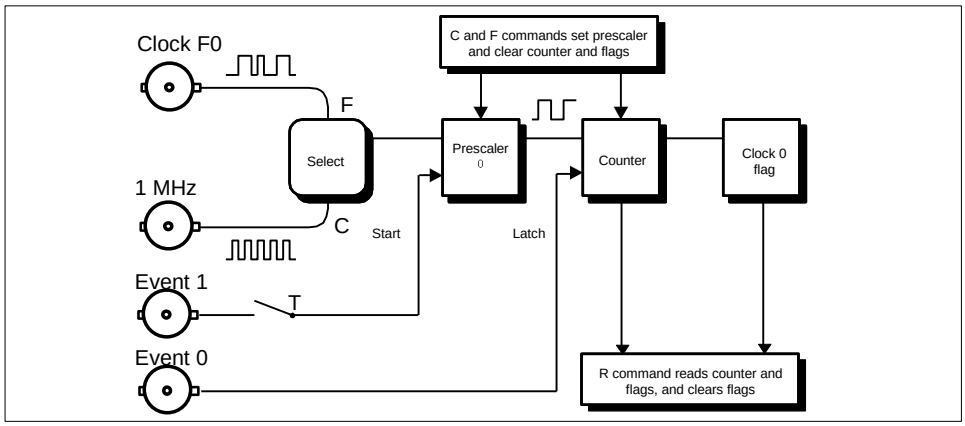
Example 9: Control of the digital port

```
Open1401                    Open communication with the 1401
DIG,S,255                   Set low 8 bits to  outputs
x=0                         Initialise the loop
repeat                      Start of the loop
       DIG,O,x              Write next value of x
       DIG,I,15             Read input bit 15 to j
       x=x+1                Increment the counter
until j=1                   Until input bit set
Close1401                   Stop communicating with 1401
```

CLKEVT Timing events with clock 0

This clock is suited to timing intervals between pulses on event inputs E1 and E0, and absolute timing of pulses on the event 0 (E0) input. Gathering and analysing arrays of times is better done by commands described in the *Event time processing* chapter later in this manual. See also the EVENT command on page 51 for a description of the properties of the event inputs, and for setting the active polarities of the event inputs.

Schematic of clock 0 as seen by CLKEVT



Clock 0 has a 16-bit prescaler that divides down a selected frequency source, followed by a 16-bit latchable counter, and a 1-bit clock flag. The input to the prescaler is either from the 1 MHz system clock (C) or from the F0 input (F) which may run at any rate to 4 MHz. The clock may be started by a pulse on the E1 input (T) or by program. The clock is latched (the counter value is copied and held until read) by a low going E0 pulse.

The clock is controlled and read by the built-in CLKEVT command. This is a sequential command, but the times are latched in hardware so data will not be lost if another command is running at the time of the event.

Command variants

`CLKEVT,clock[T],prescl`	Set up clock 0 and events
`CLKEVT,R;event1,event0,flag,count`	Read the clock counter and events

`clock` is a character, `C` to select the internal 1 MHz crystal source or `F` to select the external frequency supplied on the F0 input (at not more than 4 MHz). With all except the micro1401, H for 4 MHz and T for 10 MHz are also allowed.

`prescl` is a number, from 2 to 65535, that divides down the selected source frequency.

Set up the clock

`CLKEVT,clock[T],prescl` sets up the clock prescaler, zeros the counter, and starts the clock running immediately, if the `T` (trigger) is omitted, or sets the clock to start on receipt of a low going pulse on the Event 1 input if `T` is included. The counter will count upward from zero at a rate given by the input rate divided by the prescaler value.

Read the clock

`CLKEVT,R` returns the state of both event flags, the clock overflow flag and the counter value. If no event has been seen, or the clock has not overflowed, the respective arguments will be zero, otherwise one. The returned format is:

`event1,event0,flag,count` for example:
`1,0,0,1286`

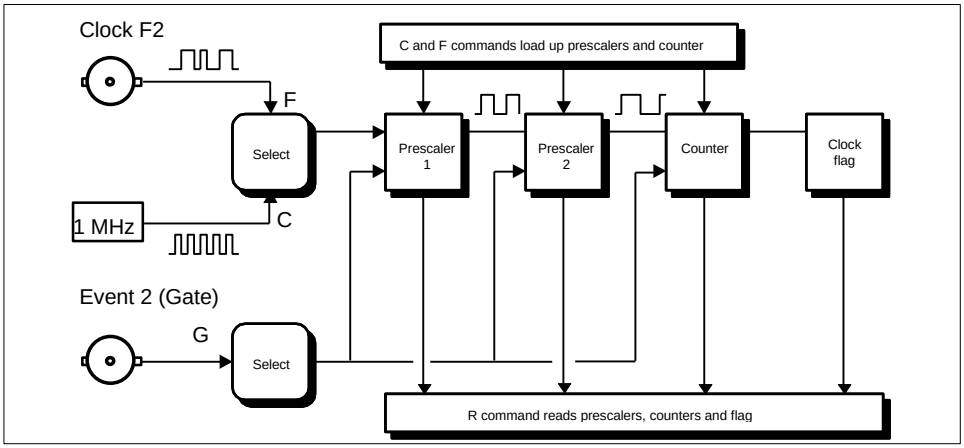
This indicates an event 1, but no event 0 or clock overflow, and a clock counter value of 1286 out of a possible 65535. The act of reading the clock will clear the clock flag and the two event flags, but the clock continues to run so further events can be seen.

TIMER2
The general purpose 48 bit clock

This built-in command controls general purpose use of clock 2. Clock 2 has four parts:

1. Clock source, either the 1 MHz clock (C) or the user input (F) of up to 4 MHz.
2. Clock Gate; a selectable hardware clock gate (G) whose function depends on the clock mode (see below). The gate is controlled by the E2 input.
3. Counter chain; 3 programmable dividers called prescaler 1, prescaler 2 and counter.
4. Out; a signal level that can control external equipment. This level also sets the clock flag which may be read by the `TIMER2,R...` command. Once the flag is set it remains set until the next `TIMER2,F...` or `TIMER2,C...` command.

Block diagram of clock 2



Command variants

`TIMER2,clock[G],mode,pre1,pre2,count` Set up the clock
`TIMER2,R,which;flag[,count[,pre2[,pre1]]]` Read the clock

`clock` `C` for the internal 1 MHz crystal, or `F` for an external TTL signal on the rear F input. All units except the micro1401 also allow `H` for 4 MHz and `T` for 10 MHz.

`mode` range 0 to 5 sets the mode of operation of the clock, see below.

`pre1` the first prescaler, the first divide down of the source, range 2-65535.

`pre2` similarly sets the second divide down, range 2-65535. The Power1 allows a special mode with `pre1=pre2=1`.

`count` sets the third stage of divide down, range 2-65535.

`which` specifies which of the three 16 bit counters to read.

The Micro2/3 and Power2/3 allow 1-65536 clock ranges and support `mode` 2 and 3 only. Modes 0, 1, 4 and 5 are for micro1401 and Power1 only and should be avoided.

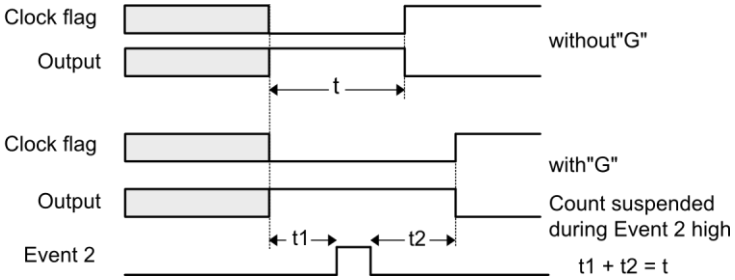
Set up the clock

`TIMER2,source...` set the clock to run from the internal clock source or from the F input. The hardware gate input (E2) is enabled by adding `G`. Clock modes are:

mode 0: Suspendable time out (obsolete)

The clock flag is cleared, the output goes high and the clock counts down from the values loaded into the prescalers and counter. When the count reaches zero, the flag is set, clock output goes low but the clock continues counting down. If the `G` code is used to enable the hardware gate, the count is suspended whenever `G` goes high. The counters and prescalers are not loaded until the gate goes low.

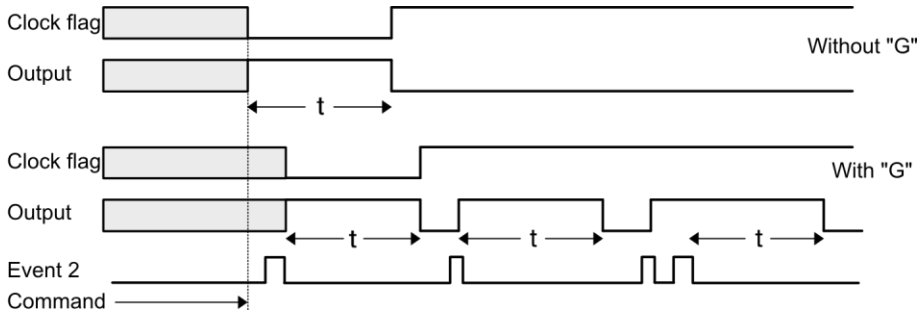
Mode 0 timing



mode 1: retriggerable one shot (obsolete)

The clock flag is cleared and output goes high and the clock will count down to zero, when output will go low and the clock continues to count. If the hardware gate is enabled, the clock will wait for a high to low transition on the gate before starting the sequence. The counters will be reset, and the sequence restarted for every high to low transition of gate (E2), even if the count is not yet exhausted.

Mode 1 timing

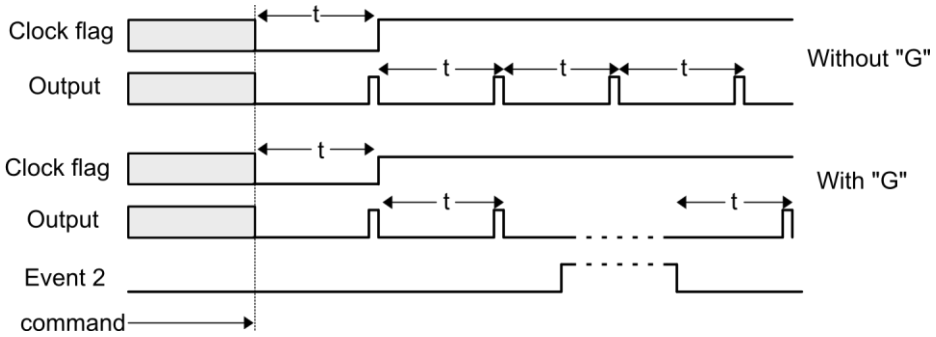


mode 2: Rate generator

In mode 2 the clock output will pulse high for 1 clock of the counter, and be low for n-1 clocks, where n is the count loaded into the counter. If the internal crystal source is used (C), the output frequency is `1000000/(pre1 * pre2 * count)`.

If the hardware gate is enabled (G), the output will be held low when the E2 input is asserted (held high). When the gate (E2) goes low, the counter and prescaler are reloaded and the sequence continues.

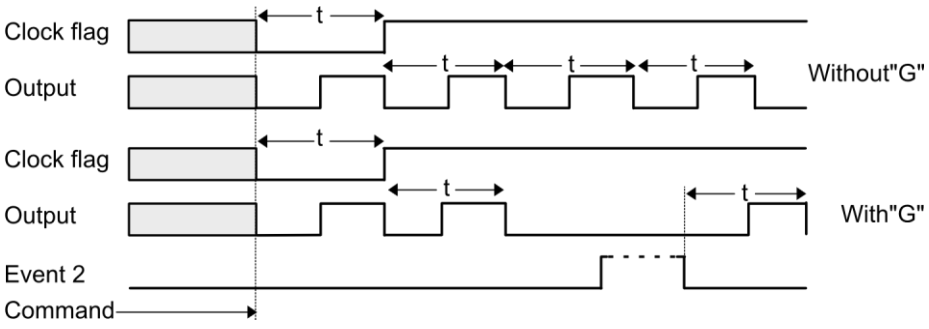
Mode 2 timing



mode 3: square wave generator

Mode 3 resembles mode 2, except that the output is low for half the count, and high for half the count if n is even (where n is the counter preset value). If n is odd, the output is low for one extra count.

Mode 3 timing



The Micro2/3 and Power2/3 support modes 2 and 3 only. In these modes, removing the gate signal suspends the count. Restoring it continues the count. The counter does not reload.

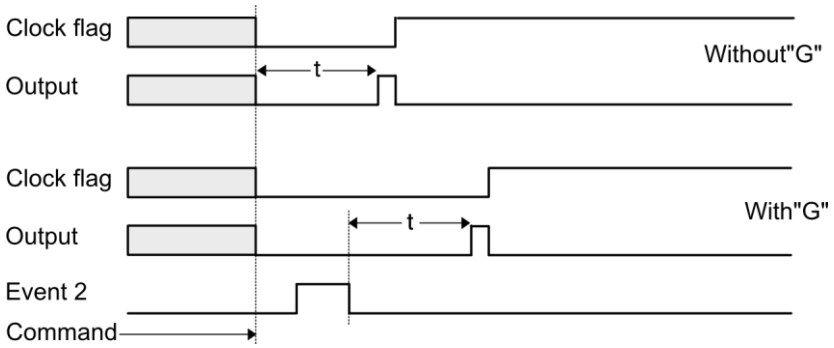
mode 4: software triggered strobe (obsolete)

Mode 4 causes the clock to produce a high going pulse of one counter clock period duration after a delay (if using 1 MHz clock) of:

`pre1 * pre2 * count` microseconds

This pulse will occur after the counter has counted from the preset value to zero. If the hardware gate is enabled, its effect is to suspend the counting when asserted (allowed to go high) and restart when pulled low.

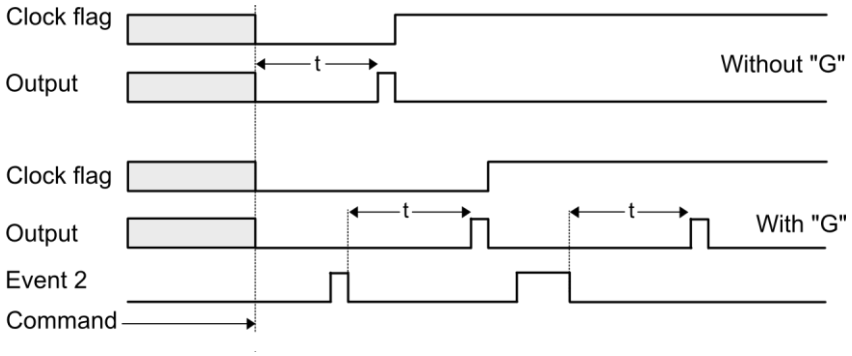
Mode 4 timing



mode 5: hardware triggered strobe (obsolete)

In mode 5, if the gate (G) is not enabled, the effect is the same as in mode 4. If the hardware gate is enabled, a high going pulse will be produced after the set delay whenever the gate input makes a high to low transition.

Mode 5 timing



Reading the clock

The `TIMER2,R` command reads the state of the clock flag, and up to three counters:

Command	Values returned
`TIMER2,R,0`	`clock flag`
`TIMER2,R,1`	`clock flag, counter`
`TIMER2,R,2`	`clock flag, counter, prescaler2`
`TIMER2,R,3`	`clock flag, counter, prescaler2, prescaler1`

This command latches the state of all three stages simultaneously. Values read from the prescalers and the counter count down to zero.

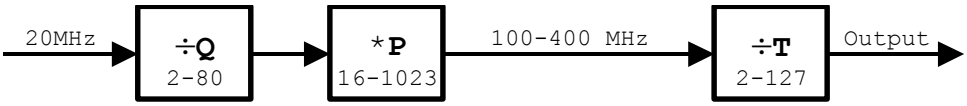
When used as a timer, `prescaler 2` holds the least significant count, `prescaler 1` the next most significant, and `counter` the most significant. The counter clock period is the time to decrement the counter by one, which is:

counter period = `pre1` * `pre2` clock periods

XFREQ Frequency synthesiser

The XFREQ command is supported by the Power1401 mk II and -3 only, and allows you to synthesise a range of frequencies that are not limited to subdivisions of 10 or 4 MHz. The frequency output can be routed to the ADC Ext input, the clock F input and to the front panel Clock Out signal (replacing clock 2 output).

The output frequency in MHz is given by `((20 / Q) * P) / T` where `Q`, `P` and `T` are integers. The range of `Q` is 2 to 80, `P` is 16 to 1023 and `T` is 2-127. The synthesiser works as in the diagram below.



The output from the multiply by P stage must lie in the frequency range 100 to 400 MHz. If the output is routed to the clock F or ADC Ext inputs, it must be no more than 10 MHz. The lowest output frequency is 100/127 MHz (0.7074016 MHz).

Command variants

`XREFQ,G,qDiv,PMult,tDiv`	Start frequency synthesis
`XFREQ,S`	Stop frequency synthesis
`XFREQ,R,output`	Route the output of the synthesiser

`qDiv` The initial divider for the 20 MHz signal in the range 2-80.

`pMult` Frequency multiplier in the range 16-1023. `P/Q` must have a value between 5 and 20 so that the output for the frequency multiplier lies in the range 100 to 400 MHz. This is the factor that allows us to generate frequencies that are not available from the other 1401 clocks.

`tDiv` The final divider, in the range 2-127.

`output` This is the sum of 1=route to ADC Ext input, 2=route to the Clock F input and 4 to route to the Clock Out BNC on the front panel, replacing any clock 2 output. If the output is used as the Clock F input, it is resynchronised to 20 MHz, which will generate a timing jitter of up to 50 nanoseconds on the derived clock output.

The default state, and after loading a command or the `CLEAR` command is equivalent to the command sequence: `XFREQ,S;XFREQ,R,0;`

A worked example

Let us suppose that we want to sample 9 channels of ADC data (non-burst mode) at 1024 Hz. We need a frequency of 9*1024 Hz. Without using the synthesiser we can get close, but we cannot manage this exactly. We want to generate a clock F frequency so that:

$9*1024 = 2*10^7 * P/(QTn)$ where `n` is the clock divider, which is the same as:
$QTn * 9 * 1024 = 2*10^7 * P$

The trick is to set `P` to take out as much of the factor on the left as we can. We can get lots of factors of 2 and 5 on the right from the 20 MHz value, so concentrate on other factors, in this case the 9, so set `P` to 9*64 = 576. We then have:

$16* QTn = 2*10^7$, or $QTn = 125*10^4$

Now `P/Q` must lie in the range 5 to 20, so the smallest allowed value of `Q` is 576/20, which is 28.8, which rounded up is 29 (28 gives P/Q > 20). Possible useful values of Q are 40, 50 or 80 (as they divide into $125*10^4$). Lets try 50. This gives:

$Tn = 25000$

Now the output frequency must be less than 10 MHz. The **`*P`** output is `20*P/Q` MHz, which is 230.4 MHz, so T must be at least 24. Setting T to 25 means that the clock divider n must be 1000. So a solution (not the only one) is Q=50, P=576, T=25 to generate a frequency of 9216000 Hz, and we set the ADC clock to use the F input and divide by 1000.

DIGTIM Sequenced digital outputs

The DIGTIM command produces a sequence of precisely timed changes of bits 8 to 15 of the digital outputs and the 1401 internal events. The internal events are equivalent to the E0, E1, E2, E3, E4 and ADC external convert inputs of the 1401, but are available under software control. You can choose to control either the digital outputs, the internal events, or both the digital outputs and internal events in parallel. See the EVENT command on page 51 for more details.

DIGTIM is multi-tasking, so can be run at the same time as other commands, and is commonly used to generate trigger signals for commands like ADCMEM, MEMDAC, PSTH and INTH as well as to control external equipment. The maximum rate of other interrupt driven commands is reduced when DIGTIM runs. You cannot use DIGTIM with the TIMER2 command, which uses the same internal hardware (clock 2) as DIGTIM.

The sequence of output changes is defined in terms of time slices. Each slice has a length that is a multiple of a basic clock period. Changes in the outputs occur at the end of the appropriate slice. We suggest that slices should not be set shorter than 10 microseconds with the micro1401 or 5 microseconds with the Micro2/3 or Power2. The Power3 will tolerate 2-3 microseconds. If other interrupt-driven commands are active, this minimum inter-slice interval must be increased. A slice length of a few times the absolute minimum is suggested. An error will be flagged when the command runs if a slice is too short.

The DIGTIM command is compatible with the DIG command, but any changes made to output bits 8 to 15 by DIG will not occur until the end of the current DIGTIM time slice.

Command variants

`DIGTIM,Ox`	Select mode of control
`DIGTIM,S[I],st,sz`	Set work table
`DIGTIM,A,mask,state,count[,jmp[,rpt]]`	Set next slice
`DIGTIM,C[G|T],preset1,preset2[,repeat]`	Set internal clock rate
`DIGTIM,F[G|T],preset1,preset2[,repeat]`	Set external clock divide
`DIGTIM,?;state,rptsdone,slice`	Get command state
`DIGTIM,S;rptsdone,slice`	Stop: returns command state
`DIGTIM,K`	Kill the command

Internal or external events

DIGTIM,Ox determines if the output sequence is to be sent to the internal events, the digital output, or both the internal events and the digital output. Parameter x is one of D, I or B. D uses the digital outputs only, I uses the internal events only and B uses both the digital output and the internal events. If the internal events are used, the corresponding external events should be disabled with the EVENT,D command. If the DIGTIM,Ox command is not given, only the digital outputs will be used. The DIGTIM,Ox command must be issued before the DIGTIM clock command starts the output sequence.

Booking memory

DIGTIM,S[I],st,sz allocates an area of memory from st as a private work space for the command. Each slice uses 16 bytes of memory so if your sequence is 5 slices long, you will need at least 5*16 = 80 bytes, so sz must be at least this. The optional I initialises the st,sz region to zeros. If I is omitted, the contents of the area are not changed.

Build the next slice

`DIGTIM,A,mask,state,count[,jmp[,rpt]]` adds a new slice to the end of the list held in the work space defined by the `DIGTIM,S` command. A 253 error will be flagged if the workspace is exceeded.

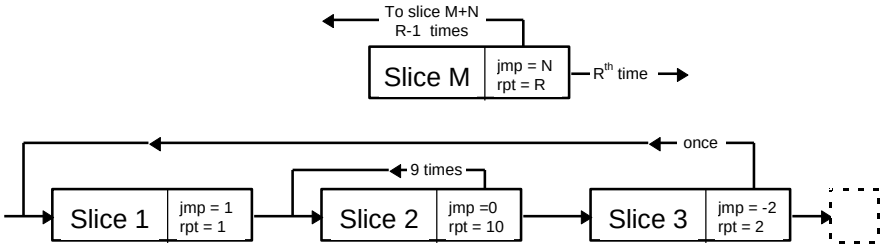
Mask values

Event	E0	E1	E2	E3	E4	Ext	-	-
Digital bit	08	09	10	11	12	13	14	15
Value in mask	1	2	4	8	16	32	64	128

`mask` determines which output bits are to be set at the end of the slice. The value to use is calculated by adding together the values from the table. For example, to select digital bit 8 or E0 set `mask` to 1. To select bit 12 or E4 set `mask` to 16. To select both, set `mask` to 17. `state` sets the values of the bits selected by `mask`. If `mask` were set to 17 and we wanted to set bit 8 high (E0) high and bit 12 (E4) low we would set `state` to 1. To set bit 12 high and bit 8 low we would set `state` to 16. Outputs not selected by `mask` are not changed.

`count` sets the length of the slice in the time units set by `DIGTIM,C...` or `DIGTIM,F...` and must be in the range 2 to 65535. The optional `jmp` (default 1) and `rpt` (default 1) arguments allow slices and groups of slices to be repeated. `jmp` sets the offset to the number of the next slice to be executed and may be positive, negative, or zero to repeat the current slice. If `jmp` is not 1, it creates a group of slices. `rpt` sets the number of times the group will be executed, and must be in the range 1 to 255 for the standard 1401 and 1 to 65535 for the others. A 253 error flags attempts to branch outside the `st,sz` region.

How DIGTIM branches are used



The slice sequence is: 1222222222231222222222223...

Setting up the clock

`DIGTIM,C` and `DIGTIM,F` commands set the basic clock period for the command and start the sequence. `C` selects the 1 MHz internal timer source, `F` selects the rear panel F input. For all 1401s except the micro1401, the `C` can be replaced by `H` for a 4 MHz clock or T for a 10 MHz clock. The `G` clock qualifier can be replaced by `T` to select an edge-trigger on E2 rather than the standard gated behaviour. With the `G` option, the clock will only run when the E2 input is held low (e.g. by shorting it to ground).

The clock source is divided down by `preset1 * preset2` to give the basic clock period. All 1401s support preset values in the range 2 to 65535. In addition, the Power1 allows both to be 1. Micro2/3 and Power2/3 also allow either or both to be 1.

The `repeat` argument (default value 1) sets the number of times the entire sequence is to be executed in the range 1 to 65535.

Checking for completion

`DIGTIM,?;state,rptsdone,slice` returns `state` as 0 if the command is finished and 1 otherwise. `rptsdone` is the number of times the entire sequence has been executed and `slice` is the currently active slice number within the sequence.

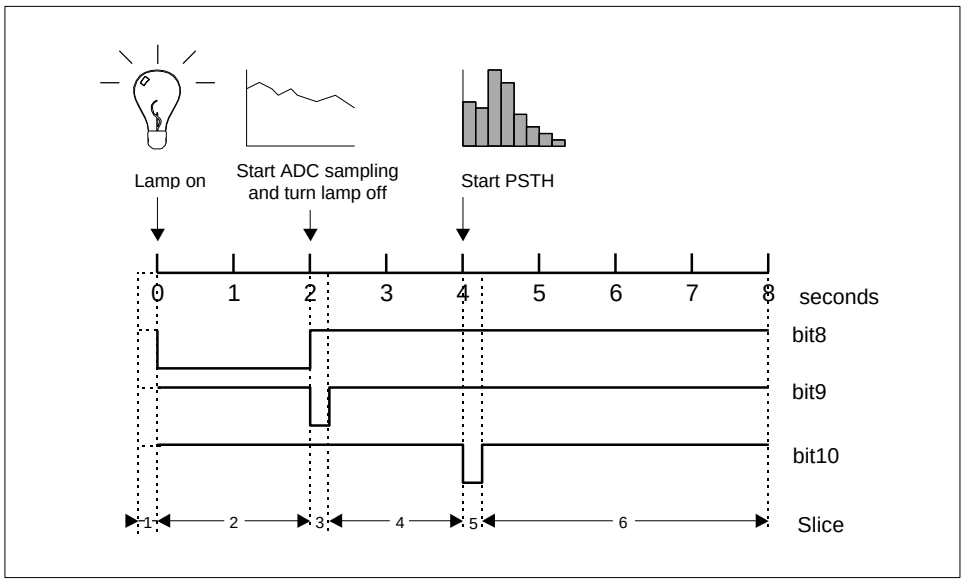
Stopping the command

`DIGTIM,S` stops the command at the end of the current slice and returns the number of repeats completed and the current slice number. `DIGTIM,K` kills the command.

DIGTIM example

This diagram shows how the DIGTIM command might be used in a simple case where an experimenter wishes to control a stimulus lamp, trigger an ADCMEM,I sampling sweep and start a PSTH command at defined times. Bits 8, 9 and 10 of the digital outputs are selected for the task. Bit 9 is connected to the E4 input to trigger the ADCMEM command, bit 10 is connected to E1 to start the PSTH command and bit 8 controls the lamp.

A typical DIGTIM sequence



The example also shows you can perform background tasks, such as set DAC outputs and read the state of the digital inputs while the DIGTIM command is controlling the digital outputs to trigger ADCMEM and PSTH.

Example 10: Sequenced digital outputs

```
DIG,O,1792                                  outputs 8, 9 and 10 high, now
ADCMEM,I,2,0,1024,0,1,CT,10,100             Set up triggered 1 kHz ADCMEM
PSTH,G,1024,1024,1,M,1                      Set up PSTH single sweep
DIGTIM,SI,2048,96                           Book space for the 6 slices
DIGTIM,A,1,0,2                  Slice 1:    2 units, bit 8 low
DIGTIM,A,3,1,20                 Slice 2:    20 units, bit 9 low, 8 high
DIGTIM,A,2,2,2                  Slice 3:    2 units, bit 9 high
DIGTIM,A,4,0,18                 Slice 4:    18 units, bit 10 low
DIGTIM,A,4,4,2                  Slice 5:    2 units, bit 10 high
DIGTIM,A,0,0,38                 Slice 6:    38 units, no output changes
DIGTIM,OD                                   Output sent to digital bits only
DIGTIM,C,100,1000                           10 Hz rate, no repeats, go
DAC,0,-32768,2                              Set analogue output 0 to - full scale
DIG,I,3                                     Read the state of digital input bit 3
DIGTIM,?                                    Wait for DIGTIM done state
DAC,0,32767,2                               Set analogue output 0 to + full scale
ADCMEM,?                                    Repeat until zero is returned
PSTH,?                                      Repeat until 'status' = 0
TOHOST,0,2048,data                          Transfer ADC and PSTH data to host
```

This diagram shows the sequence of messages and data transfers between the 1401 and the host computer during the example.

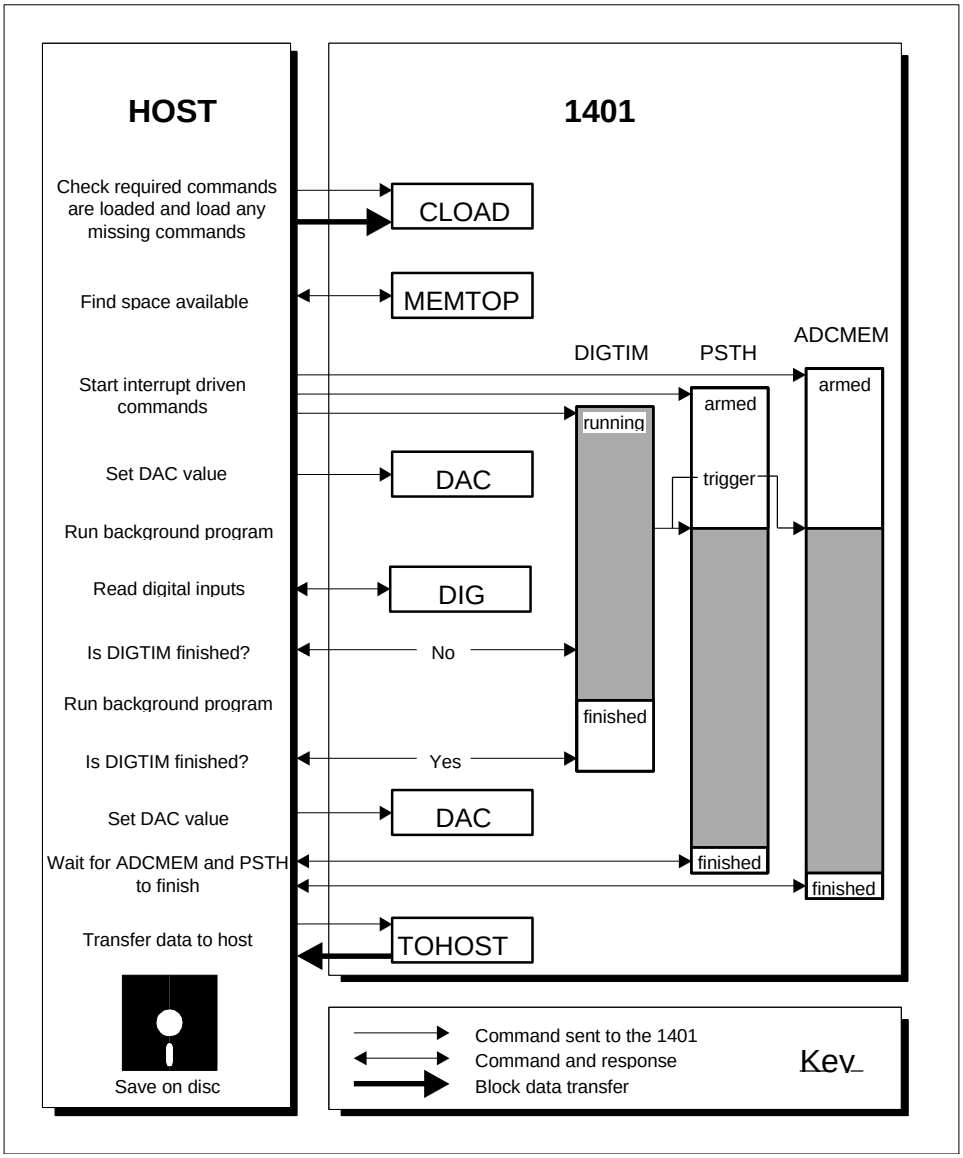
Note that this example requires external wiring between digital output bit 9 and the E4 front panel input (to trigger ADCMEM) and digital output bit 10 and the E1 input (to trigger

PSTH). The sequence could be made to drive the two triggers using the internal events by making the following changes:

1. Use DIGTIM,OB in place of DIGTIM,OD
2. Use bit 14 or 15 for the lamp to avoid bit 8 (E0) interfering with the PSTH command.
3. Use bit 12 (E4) in place of bit 9 to trigger ADCMEM directly.
4. Use bit 9 (E1) in place of bit 10 to trigger PSTH.

A version of the example 10 with these modifications is provided on the language support disk as example 10a.

Activities in the 1401 and host for this example



EVENT
The 1401 internal events

The external event inputs E0, E1, E2, E3, E4 and ADC external convert are often used to start commands on a pulse. It is also useful for DIGTIM to be able to control these inputs, without external wiring to link digital outputs to the event inputs, so they are mimicked by a set of 'internal events' that can be set by software with the EVENT command.

You can use internal and external events together or disable external events for internal use only. The command also sets the active polarities of the external event inputs and routes the Trigger, Event 0 and 1 inputs. Any changes made by EVENT are undone by CLEAR, RESET (hardware or software), power up and on loading any command.

Command variants

EVENT,P,select	Set external event polarity
EVENT,D,select	Disable external events
EVENT,I,select	Set internal event state
EVENT,M,mode	Set internal event mode
EVENT,T,select	Link selected inputs to front panel Trigger (micro/Power)
EVENT,E,select	Connect selected signals to front panel Event 0 and 1 (micro/Power)
EVENT,R,select	Connect selected inputs to the rear panel inputs (micro/Power)
EVENT,X,outsel	Power2/3 only: select signal on the rear panel event pin 8
EVENT,L,select	Power3 only: connect front panel E0/E1 to digital input bits 0/1

`select` Sets which events the command affects. Each event has a value, given in the table below, and `select` is the sum of the values of the selected events. Digital input bits 8 and 9 values are for the EVENT,E and EVENT,R commands.

Event	E0	E1	E2	E3	E4	ADC Ext	(Dig 8 & 9)
value	1	2	4	8	16	32	(192)

`mode` A value used to set the internal event mode. This is the sum of two values:

mode	clocked	pulsed
value	64	128

Clocked mode causes any change of value written to the internal event to be suppressed until the next time the clock 2 flag sets. The clock 2 flag must be cleared before it can be used to clock the internal events again (for example by using the TIMER2 command to restart clock 2).

Pulsed mode causes a pulse on internal events selected by EVENT,I,select. If pulsed mode is not selected, the value written to the event sets the state.

`outsel` Set 1 for 4 MHz on the rear panel event connector pin 8, 0 for no output. Other values of outsel are reserved. By default, nothing is connected to the output.

Set the active event polarities

EVENT,P,select controls the active state of the inputs. The normal event input polarity is set by a switch pack (see the *Owners handbook*). The CED recommended state is with the inputs active low. The inputs will go high with nothing connected, so if the inputs are set for positive/high as the active state unconnected inputs will appear active.

The event polarity set by the switch is overridden with this command. Selecting an event gives it negative/low active state (recommended) otherwise they have positive/high active states. To set event 2 to have a high active state and the rest to be active low:

```
EVENT,P,59
```

59 = 1 + 2 + 8 + 16 + 32 selecting events 0, 1, 3, 4 and external convert. Set the event polarity before other operations as changes cause edges to be seen by the event inputs.

Disable the events

EVENT,D,select disables selected external events allowing the internal event register to control a process without interference from external pulses.

Set the internal event state

`EVENT,I,select` sets internal events according to `select`. Selecting an event will set the corresponding internal event to active (in level mode) or cause a pulse (in pulse mode) if the corresponding external event is disabled, or can be guaranteed to be permanently in the non-active state. Unselected events are set to the non-active state in level mode and are not changed in pulse mode. Selecting an event that is connected to the external events can mask further changes and should be avoided.

Set the internal event mode

`EVENT,M,mode` sets the internal event mode, defined in the table above.

Link to Trigger input

`EVENT,T,select` connects the Trigger front panel input to the selected inputs. Any or all of E0, E1, E2, E3, E4 and ADC Ext convert inputs can be connected.

Link to Event inputs

`EVENT,E,select` selects the source of the E0 and E1 inputs and digital input bits 8 and 9 as the front panel Event 0 and Event 1 inputs. The value of `select` is the sum of:

Value	Connection
1	Connect E0 input to front panel Event 0
2	Connect E1 input to front panel Event 1
192	Connect digital input 8 to front panel Event 0 and digital input 9 to front panel Event 1. You cannot connect only one digital bit.

Link to rear inputs

`EVENT,R,select` connects the selected inputs to the rear panel signal sources. There are separate inputs for the E0, E1, E2, E3, E4 and ADC Ext convert inputs on the rear panel Event connector, and for the digital input bits 8 and 9 on the digital input connector. See the *Owners manual* for details.

Event out

`EVENT,X,enable` is available with the Power2 and 3 to enable (1) or disable (0) a 4 MHz output from the rear panel event connector, pin 8. As this signal has the potential to cause electrical interference, it requires a board jumper (normally not fitted) to enable it.

Digital inputs bit 0 and 1 source

`EVENT,L,select` is available with the Power3 to connect the E0 and/or the E1 front panel inputs to digital inputs 0 and/or 1. This is for use in Spike2 to provide digital inputs on BNC connections.

Example of use

Consider the following situation. An experimenter wishes to play out an analogue waveform as a stimulus, and start logging an analogue waveform and also capture digital events, with the start of all three activities synchronised. The following commands could be sent to the 1401 (we assume the `ADCMEM`, `MEMDAC` and `PSTH` commands are loaded):

Example 11: The EVENT command

```
EVENT,D,26;                                  external events 1, 3 and 4 disabled
EVENT,M,128;                                 set pulsed internal event mode
MEMDAC,I,2,0,200,0,1,CT,100,10;              play data at 1 kHz
ADCMEM,I,2,200,200,0,1,CT,100,10;            record data at 1 kHz
PSTH,G,400,200,1,M,1;                        start PSTH data capture

All the commands are now waiting for triggers. PSTH is waiting for event 1, ADCMEM for
event 4 and MEMDAC for event 3. We can now start all the commands together:

EVENT,I,26;                                  cause all events to pulse
```

Array arithmetic

Introduction to array arithmetic

The array arithmetic description is split into two chapters: this one, on simple array arithmetic operations, and the following chapter on the more complex FFT-based operations. If any of these commands fit the task in hand, they are probably well worth using because a) the 1401can be faster than typical high level languages in the host, and b) it is often possible to reduce the volume of data to be sent back to the host, freeing it for other system jobs.

The commands run as background tasks in the 1401 and can be overlapped with data acquisition. The timing figures given in this chapter and the next one must be seen as approximate; some depend significantly on the data. The rates will be slower if other commands are running, diluting the processor effort.

There are 8 array arithmetic commands, divided into two related groups of four. Each of the four command families is available in the `XX2` form, for handling 16 bit data, and the `XX1` form for 8 bit work. The `XX1` family is rarely used and is not optimised for speed in the same way as the `XX2` commands. The four families are:

Command	Function	page
`SS1,SS2`	Single array manipulation, e.g. find the biggest element	54
`SD1,SD2`	Double precision arithmetic, and double/single conversion	56
`SM1,SM2`	Multiple array manipulation e.g. array multiplication	58
`SN1,SN2`	Separation and interleaving of channels	59

Important restriction

All the array arithmetic code assumes that arrays are aligned in memory to boundaries that are multiples of the size of the item in the array. This means that arrays of single precision (16-bit) data must start at an even address (2*n) and arrays of double precision data (32-bit) must start at an address that divides by 4 with no remainder (4*n). There is no restriction for 8-bit data.

This is rarely a problem for 16-bit data, even with code written for older versions of the 1401. However, you will occasionally find legacy code that starts a double precision array at an address that is 4*n + 2. This will give an argument error.

This restriction is due to the way that the ARM[tm] and StrongARM[tm] processor accesses memory. Although we could arrange to read 16 and 32-bit data at any alignment, it would be very much slower.

This restriction also applies to data capture routines that store data into arrays.

SS1 and SS2 Single array commands

The single array commands carry out fast array manipulation on single arrays in the user data space of 1401.

The arrays are referred to by a start address in bytes, and by a byte size, which is the number of bytes in the data area to be used. 1401 will automatically check that the area requested does not run over the top of available memory. The `SS1` commands take less space, but the data range is +127 to -128, not +32767 to -32768.

Individual commands

There are thirteen different sub-commands available in both the `SS2` and `SS1` commands. These are listed below, with a brief description, and the execution time in microseconds per element for `SS2`. `SS1` times are similar.

Command	Operation	micro	Micro2	Micro3	Pwr1	Pwr2	Pwr3
`SSx,N,st,sz`	Negate array	1.40	0.20	0.19	0.059	.0138	.0120
`SSx,M,st,sz`	Modulus of array	1.10	0.20	0.17	0.058	.0150	.0121
`SSx,C,st,sz,arg`	Set array to Constant arg	0.10	.034	.022	0.020	.0038	.0013
`SSx,+,st,sz,arg`	Array = array + arg	0.54	0.13	0.12	0.041	.0150	.0085
`SSx,S,st,sz,shift`	Scale array by 2^{shift}	1.44	0.20	0.19	0.059	.0137	.0122
`SSx,B,st,sz;size,pos`	Find Biggest in array	0.74	0.17	0.14	0.054	.0122	.0102
`SSx,L,st,sz;size,pos`	Find Least in array	0.74	0.17	0.14	0.054	.0121	.0101
`SSx,D,st,sz`	Differentiate array	1.33	0.18	0.18	0.055	.0128	.0103
`SSx,I,st,sz[,shift]`	Integrate array	1.48	0.21	0.20	0.063	.0138	.0112
`SSx,F,st,sz`	Butterworth Filter array	1.68	0.28	0.24	0.084	.0205	.0150
`SSx,R,st,sz`	Time Reversed filter	1.68	0.28	0.24	0.084	.0205	.0150
`SSx,*,st,sz,arg,shift`	Array *= arg / 2^{shift}	1.87	0.23	0.20	0.065	0144	.0104
`SSx,A,st,sz;value`	Average array value	0.74	0.17	0.14	0.053	.0114	.0090

In the individual descriptions, `st` and `sz` refer to the array start and array size as a byte address, and a byte size. `SS2` commands operate on word (2 byte) elements so an `SS2` array with 100 word elements would have `sz` = 200.

Negate array

`SSx,N,st,sz` The array is replaced with its negated values. In the standard 1401 a value of full scale negative ($8000 or -32768) will appear unchanged (+32768 is not a valid 2's complement number). In all other 1401s, -32768 becomes +32767.

Replace by modulus

`SSx,M,st,sz` The array is replaced with its modulus. Beware that full scale negative will remain as full scale negative in the standard 1401.

Set to constant

`SSx,C,st,sz,arg` All elements of the array are set to the value of `arg`.

Add constant value

`SSx,+,st,sz,arg` The array has the value of `arg` added to it. No warning is given in the case of an overflow. The argument may be positive or negative.

Shift the array

`SSx,S,st,sz,shift` The array is multiplied by 2^{shift}, where `shift` is in the range -15 to +15. For the standard 1401 only, the time taken depends strongly on the number of shifts. The result of this operation is only defined for `shift` in the specified range. No warning is given if the result of an operation exceeds 16 bits. No rounding is done, only pure shifts, so the result of shifting a negative number to make it smaller will never be 0, but tends to -1.

Find biggest and smallest element

`SSx,B,st,sz;size,pos` This command will search the array for the maximum value and return the value of the maximum, and its word offset into the array. An offset of zero corresponds to the first two byte word. `pos` is a byte offset for `SS1`.

`SSx,L,st,sz;size,pos` This command returns the most negative value found in the array, and its position.

Differentiate array

`SSx,D,st,sz` The array is replaced by the simple differences between each element. No warning is given of overflows. The first element is always replaced by 0.

Integrate array

`SSx,I,st,sz[,shift]` The array is replaced by its integral, optionally divided by 2^{shift}, where shift is in the range 0 to 15. Integration is very prone to overflow unless it is scaled. Each element is replaced by the sum of all the elements up to itself. `SSx,I` is the inverse of `SSx,D` above. `shift` defaults to 0 if omitted.

Forwards and reverse time filter

`SSx,F,st,sz` The array is filtered by a second order Butterworth filter with a cut-off at about 10 times the point spacing. The filter processes the data in ascending order of addresses. You should be aware that the filter will shift any peak in the data to the right.

`SSx,R,st,sz` The same filter as above but run backwards in time. This can be useful where multiple filter passes are needed and the time delays introduced by filtering need to be minimised.

Multiply array

`SSx,*,st,sz,arg,shift` Each array element is multiplied by variable `arg` to 32 bit precision, then the result is divided by 2^{shift}. No warning is given of overflow.

Average value

`SSx,A,st,sz;value` The average value of the array is returned as an integer; the array is not changed.

Use with 8 bit data

The descriptions above are given for `SS2`, with 16 bits. For the `SS1` type, shift ranges of 0 to 15, or -15 to +15 become 0 to 7, and -7 to +7 and -32768 becomes -128. Word offsets in `SSx,B` and `L` are replaced by byte offsets.

SD1 and SD2 Double precision array commands

These commands offer double precision arithmetic, and the conversion between double precision and single precision arrays. In this context, for SD2, double precision means 32 bit signed arithmetic and single precision is 16 bit signed arithmetic. SD1 similarly processes 16 and 8 bit arrays.

The command has the general form:

SDx,fn,dp,sp or dp,sz[,arg]

dp start of a double precision array

sp start of a single precision array

sz size in bytes of the double precision array. If there are to be n data items in the double precision array, then sz must be set to 4 * n (SD2) or 2 * n (SD1).

arg an extra value required by some functions

fn Two characters which define the function required.

Timings for the double precision arithmetic commands in microseconds per element for SD2. Timings for SD1 are similar.

Command	Operation	micro	micro2	Micro3	Power	Pwr2	Pwr3
SD2,S+,dp,sp,sz	Add single to double	1.29	0.292	0.252	0.106	0.035	0.020
SD2,D+,dp1,dp2,sz	Add double to double	1.29	0.288	0.283	0.124	0.044	0.022
SD2,D/,dp,sp,sz,arg	single = double/arg	8.23	1.60	1.32	0.452	0.128	0.084
SD2,DS,dp,sp,sz;scl	scales dp to sp array	2.12	0.436	0.424	0.143	0.060	0.033

Add single precision array to double

SD2,S+,dp,sp,sz Adds the single precision 16 bit array of size sz/2 bytes to the double precision array of size sz bytes. Each array is sz/4 data elements long.

Add double precision array to double

SDx,D+,dp1,dp2,sz Adds the double precision array 2 to double precision array 1, both of size sz bytes. The number of data elements in each array is sz/4.

Divide double by constant

SDx,D/,dp,sp,sz,arg Divides the double precision array Divide double precision array by constant; by the integer arg and puts the result in the single precision array. The number of data elements in each array is sz/4.

Scale double to single

SDx,DS,dp,sp,sz;scl The single precision array is set to the best representation of the double precision array such that:

sp array * 2^{scl} =: dp array

Where =: means is as nearly equal as the word size allows. The scale factor needed is returned to the host as scl, which is always positive so if an array already lies within the signed integer range, no scaling will be done and scale will be returned as 0.

Example: Double precision average

ADC channel 0 is sampled in response to a trigger at 10 kHz using the ADCMEM command. The incoming signal is accumulated to double precision and the double precision total is divided by the number of sweeps, both using the SD2 command, to give a constant amplitude average.

Example 12: Double precision average

Load the SS2, SD2 and ADCMEM commands, if necessary.
Put up a message to show the demonstration has started.
Ask the user to enter values for the number of data points (between 10 and 1024), and sampling rate (15 Hz to 10 kHz).
Calculate 1401 sampling array size = data points * 2
Calculate clock count = 500000/rate, assuming preset = 2
Ask the user for the number of sweeps (must be positive!)
Open communications with 1401

```
SS2,C,0,4*size,0                                  Clear data arrays in 1401

Start a loop, repeated for the chosen number of sweeps
ADCMEM,I,2,0,size,0,1,CT,2,count                  Sample data
ADCMEM,?                          Request sampling status, wait for done
SD2,S+,size,0,dsize                       Add raw data to accumulator
SD2,D/,size,3*size,dsize,i                                Form average
End of data capture loop

When the specified number of sweeps is done, say 'Finished'
```

SM2 and SM1 Multiple array manipulation

There are the usual two versions of these commands; `SM2` for 16 bit and `SM1` for 8 bit operations. This command is used to manipulate a pair of arrays of data. The general form is:

```
SMn,x,dest,srce,size[,arg]
```

`dest` is the start address (as a byte offset from the start of the user data area) of the result array.

`srce` is the start address of the source array.

`sz` is the size in bytes of both arrays. Remember that the `SM2` command deals in 16 bit words, so the size must be twice the number of words in each array, for `SM1` `sz` is plain bytes.

`arg` is an additional argument required by some commands.

`x` is a character defining the operation required.

This table shows typical timings for the `SM2` commands in microseconds per element. The `SM1` timings are similar to the `SM2` timings.

Command	Operation	micro	Micro2	Micro3	Power	Pwr2	Pwr3
`SM2,C,dest,srce,sz`	Copy source to dest	0.20	0.070	0.063	0.037	0.020	0.006
`SM2,+,dest,srce,sz`	Dest = dest + source	1.58	0.218	0.203	0.075	0.025	0.015
`SM2,-,dest,srce,sz`	Dest = dest - source	1.73	0.288	0.203	0.075	0.025	0.014
`SM2,*,dest,srce,sz,sh`	Dest = dest * source/2^{sh}	2.45	0.288	0.250	0.091	0.025	0.020
`SM2,/,dest,srce,sz,sh`	Dest = dest * 2^{sh}/source	7.85	1.53	1.32	0.475	0.161	0.113
`SM2,X,dest,srce,sz`	eXchange source, dest	2.17	0.218	0.321	0.071	0.027	0.017

Copy an array `SM2,C,dest,srce,sz` Copy the source array to the destination array. Arrays may overlap; the 1401 deals with this correctly.

Add two arrays `SM2,+,dest,srce,sz` Add the source array to the destination array. If the arrays overlap, a meaningful result will only be obtained if the source array starts higher in memory than the destination array.

Subtract an array `SM2,-,dest,srce,sz` Subtract the source array from the destination array. If the two arrays overlap, the result will only be meaningful if the source array starts higher in memory than the destination array.

Exchange two arrays `SM2,X,dest,srce,sz` Exchange the two arrays; they may NOT overlap.

Multiply two arrays `SM2,*,dest,srce,sz,sh` The destination is multiplied by the source, and the result divided by 2^{sh}. The shift count must be positive.

Divide one array by another `SM2,/,dest,srce,sz,sh` The destination is multiplied by 2^{sh}, and the result is divided by the source. The shift count must be positive.

SN1 and SN2 Extract, interleave and separate

These commands are provided to interleave and un-interleave data for the waveform input and output commands MEMDAC, ADCMEM, PERI32 and ADCBST. SN2 is for 16 bit arrays, SN1 is for 8 bits. When these commands transfer multi-channel data, it is interleaved to give the fastest possible transfer rates. For example, consider writing to DAC channels 0, 2 and 3. To get the required output, the data in 1401 memory must be:

Channel	0	2	3	0	2	3	0	2	3	0	2	3	...
Data	4	99	20	4	98	21	4	97	22	4	96	23	...

and will be output as:

Channel	**Data**				
0	4	4	4	4	...
2	99	98	97	96	...
3	20	21	22	23	...

Further consider the case where ADC channels 0, 1, 2, 3, 4 and 5 are sampled using the ADCMEM command. The data will appear in memory as:

Channel	0	1	2	3	4	5	0	1	2	3	4	5	...
Data	0	20	98	-1	50	30	1	21	96	-3	52	33	...

Although this is an efficient way for the computer to store the data, humans find this hard to work with. There are three sub-commands to help with this situation. The times below are shown in microseconds per array element for SN2.

Command	Operation	micro	Micro2	Micro3	Power	Pwr2	Pwr3
SN2,X,dest,srce,sz,int	eXtract one channel	1.33	0.171	0.171	0.071	0.040	0.016
SN2,S,dest,srce,sz,int	Separate interleaved	1.40	0.171	0.164	0.075	0.029	0.016
SN2,I,dest,srce,sz,int	Interleave separated	1.28	0.171	0.141	0.056	0.022	0.013

`dest` The start of the destination (or result) array.

`srce` The start of the source array, the data to be worked on. The destination and source arrays must not be the same, or the data arrays will be scrambled.

`sz` The size in bytes of the destination array. The convention in 1401 commands is that sizes are in bytes, and are always of the areas to be written to.

`int` The number of channels to be interleaved, separated, or extracted from. This number must be in the range 1 - 255, though 1 is not going to do anything very useful, as it will just copy the source to the destination in all cases!

In the descriptions which follow, we assume that `int` is set to 3 and we represent the data for each channel by `a`, `b` or `c`. Thus a set of data for channel `a` will be represented as:

```
a a a a a a a a a a
```

being 10 points for channel `a`. We further assume that each channel is 10 data values long. Thus we are always dealing with either three contiguous arrays:

```
a a a a a a a a a a b b b b b b b b b b c c c c c c c c c c
```

or one interleaved array:

```
a b c a b c a b c a b c a b c a b c a b c a b c a b c a b c
```

Of course, this doesn't mean you are restricted to this set of numbers; the arrays may be any size that will fit in 1401 memory.

Extract one channel

`SN2,X,dest,srce,size,int` Take the data for one channel from source, which is `sz` * `int` bytes long, and transfer it to the `dest` array which is `sz` bytes long. This is expressed as:

source `a b c a b c a b c a b c a b c a b c a b c a b c a b c a b c`
dest `a a a a a a a a a a`

To extract the b channel, you must add 2 bytes per data item (`SN2`) to the source address (1 byte for `SN1`) so the source array looks like:

source `b c a b c a b c a b c a b c a b c a b c a b c a b c a b c x`

and so on (`x` stands for data beyond the original array).

Separate interleaved channels

`SN2,S,dest,srce,sz,int` This command operates on a source array of interleaved data such as that gathered by `ADCMEM`, or data prepared for `MEMDAC`. The data is changed from `sz` points of interleaved data to a series of contiguous arrays, one for each channel:

source `a b c a b c a b c a b c a b c a b c a b c a b c a b c a b c`
dest `a a a a a a a a a a b b b b b b b b b c c c c c c c c c c`

The `sz` parameter must be a multiple of `int` words long, or error 253 will be flagged (error in command execution), and no separating will be done.

Interleave channels

`SN2,I,dest,srce,sz,int;` This is the logical inverse of the separate command. It takes a series of contiguous arrays, and interleaves them to give an array which is suitable for use by `MEMDAC`. The same restrictions apply as for the `SN2,S,...` command.

source `a a a a a a a a a a b b b b b b b b b c c c c c c c c c c`
dest `a b c a b c a b c a b c a b c a b c a b c a b c a b c a b c`

This section is intended for those who wish to use the FFT-based family of commands to perform spectral analysis on data arrays, or who wish to produce waveforms with a given spectral content. It is intended to help you to get useful results out of the software with the correct scale factors but does not pretend to teach you the theory of the Fast Fourier Transform (FFT).

The family contains four commands:

`FFT` This command will perform either a forward or a reverse FFT on an array of 16 bit data stored in the 1401 memory. The command returns a scale factor to the host computer that indicates the number of times the result has been divided by 2 to avoid overflow. Note also that the forward transform is returned too large by a factor of the number of data points entered. This FFT assumes a purely real time domain array.

`GAINPH` The command takes the result of a forward FFT which must be flipped (see below in full descriptions for meaning of flipped) and converts the first half of the array into numbers proportional to the log power of the spectrum, and the second half to numbers proportional to the relative phase of the spectrum.

`ADDPWR` This command takes the result of the forward FFT and produces a double precision array proportional to the power and adds this array into a double precision buffer. This command is used for spectral averaging.

`DLOGPWR` This takes the result of a series of calls to `ADDPWR` and makes a single precision array proportional to the (average) log power of the data.

In the standard 1401 these commands must be loaded separately; for all other 1401s the `FFT` command includes the other three. If you use the language support `Ld` function to load these commands, specify `FFT` first to ensure compatibility.

Timings for the commands

size	micro	Micro2	Micro3	Power	Pwr2	Pwr3
32	0.5	0.075	0.057	0.017	0.005	0.004
64	1.1	0.169	0.125	0.038	0.010	0.009
128	2.5	0.376	0.272	0.083	0.021	0.020
256	5.4	0.829	0.591	0.182	0.047	0.043
512	11.8	1.81	1.28	0.398	0.101	0.095
1024	25.8	3.94	2.74	0.860	0.219	0.207
2048	55.9	8.57	5.92	1.854	0.472	0.447
4096	120	19.4	13.5	3.984	1.000	0.960

The FFT command can be used for arrays that are from 8 up to 4096 points; the sizes must all be a power of two. The times are given in milliseconds. Times are not linearly related to array size but are proportional to $n*Log_2(n)$ where `n` is the transform size.

Related command timings

	micro	Micro2	Micro3	Power	Pwr2	Pwr3
GAINPH	6.4	1.14	0.887	0.298	0.079	0.074
ADDPWR	1.63	0.197	0.149	0.043	0.011	0.011
DLOGPWR	3.14	0.584	0.446	0.147	0.021	0.019

This table gives the time required, in milliseconds, for the FFT related functions to process the result of a 1024 point FFT. The time taken scales linearly, so a 2048 point FFT takes double the time and a 512 point FFT takes half the time.

FFT The Fast Fourier Transform

The two basic functions offered by the FFT command are the forward transform `FFT,F...` and `FFT,I...`, the inverse transform. The forward transform converts a series of data points, for example a waveform, into a representation based on the frequency content of the waveform. The inverse FFT carries out the reverse, converting a frequency representation into a time series. There are two variations of these commands in which the second half of the data is naturally ordered or 'flipped'.

The inverse transform

The inverse transform works on data that represents the amplitudes of sine and cosine waves which are to be added together to produce the resultant waveform. The length of the data array must be a power of 2 because of the algorithm used to transform the data quickly. The lowest number of points allowed is 8. The transform is of 16-bit data, so the `sz` parameter above must be twice the number of data points. The data array can be thought of as being in two halves, the first half representing the amplitudes of the cosine waves to be accumulated, and the second half the amplitudes of the sine waves. The transformation is 'in place'; the result occupies the same space as the data.

`FFT,I,st,sz;scale`	Inverse FFT
`FFT,IF,st,sz;scale`	Inverse pre-flipped FFT

Input data first half

The first point represents the amplitude of a cosine wave of zero frequency, i.e. a constant value to be added to the result. This value is twice the average value of the resultant waveform. To set your result to the constant value 10, you must set this to 20.

The second point holds the amplitude of a cosine wave of frequency 'one'. If there are 1024 data points then setting the second point in the array to 1000 will cause a cosine wave of amplitude 1000 to be added into the result, and this cosine wave will take 1024 points to complete one cycle.

The third point holds the amplitude of a cosine wave of frequency 'two'; there are two cycles in the data points. The nth. point holds the amplitude of a cosine wave of frequency 'n-1'. It will take n-1 cycles to fill the data space.

Input data second half

The second half of the data array holds the 'sine' components of the data. This is a little more complicated because the `FFT,I,st,sz` command expects the data in a strange order. We will consider the `FFT,IF,st,sz` command to start with because the data order is simpler, and come back to the non-flipped command. Within the second half of the data, the contents are very much as for the first half, but where we considered cosine waves above, we now consider sine waves. There is another difference. The first point in the second half does not correspond to the sine wave component of the zero frequency (which would be identically zero), but instead refers to the cosine wave component of the Nyquist frequency, half the sampling rate. As with the DC component, you must set the Nyquist value to twice the amplitude of the required output cosine wave.

We see that if we have `M` data points as input to the inverse FFT (`FFT,IF,st,sz`) the first `M/2 + 1` points are the cosine components and the second `M/2 - 1` points are the sine components. These are sometimes referred to as the real and imaginary parts. The FFT itself uses the data in a different order; the second half of the data, except for the first point in the second half, is backwards. The addition of the extra `F` in the command tells the FFT command that the data is in the 'human understandable' format and must be 'flipped'.

The inverse transform returns a scale factor. This is the number of times that the result was reduced by a factor of 2 during the transform. The true result is given by:

true result = actual result * 2^{scale}

The forward transform

The forward transform does the opposite of the inverse transform; it extracts the spectral content of a sampled waveform. However caution must be used when interpreting the

results. There are problems associated with the forward FFT, some of which are intrinsic to the transform itself, and others that are caused by the manner in which it is used.

`FFT,F,st,sz;scale`	Forward FFT
`FFT,FF,st,sz;scale`	Forward FFT with flipped result

Consider the most common use of the FFT, which is to obtain the spectrum of a sampled analogue waveform. The first problem is to select the waveform sampling frequency. The FFT will give a transform which gives frequencies from zero to half the sampling frequency. This result will be of no practical use at all unless there are no components in the original data at or above half the sampling frequency. This critical sampling rate is called the Nyquist frequency. If there is energy at frequencies above the Nyquist rate, it will be impossible to distinguish it from lower frequency signals. Once the signal is sampled it is TOO LATE; the signals must be band limited before they are sampled.

The second problem is that the FFT mathematics assumes that the data for transformation is repeated indefinitely in time. If the start and end of the data do not join smoothly the transform 'sees' this discontinuity as part of the signal. This discontinuity has energy at unexpected frequencies.

The usual solution to this problem is to multiply the data sample by a 'window' - another array which is small at the ends and 1 in the middle. This makes the discontinuities become less important. Use of a window will obviously throw away some of the information near the ends of the sample but the result is so much better than not windowing that it is usually preferred. 'No windowing' really implies use of a rectangular window, unity during the sample and zero before/after. Many windows have been proposed, optimised for varying features; the raised cosine is a good choice for general purpose work. The example program generates and uses such a window. For a more detailed treatment see e.g. Rabiner & Gold, *Theory and Application of Digital Signal Processing*, (Prentice-Hall).

The third problem is that the FFT of a sampled waveform is only an approximation to the Fourier transform of a continuous signal. It is more closely akin to the Fourier series expansion of a function between some limits. Consider the result of the FFT; it has only values defined at discrete frequencies corresponding to the fact that the data was sampled at discrete times. Suppose these frequencies are 10 Hz apart. What happens to a frequency component at say 105 Hz?

There is no exact point in the transform to correspond with this data. The answer is that this signal appears at a reduced level at 100 Hz and at 110 Hz (the amplitude is reduced by about 0.64). This problem is not as severe as it sounds because the smearing produced by any practical window will generally smooth out irregularities caused by this effect.

So we have some important rules to apply to the use of the forward FFT:

- The sampled data must have been filtered to remove components at the Nyquist frequency (half the sampling rate) and higher. It is usually good practice to sample at least 4 times faster than the highest frequency.
- The sampled data must be windowed to reduce the effect of spectral smearing.
- Remember that the result of a FFT is a spectral estimate and generally is not the same as the output of a perfect spectrum analyser. The FFT operation will introduce some numerical noise to any processed signal, such that a 1024 point transform on perfect 12 bit data can degrade in the worst case by 1 to 2 bits.

Use of the forward transform

`FFT,FF,st,sz;scale` This command performs a forward transform on the 16 bit data starting at `st` and of length `sz` bytes. The number of points (`sz`/2) must be a power of 2 between 8 and 4096. The command returns a scale factor, being the number of times the

result has been divided by two during the transform. The result is returned too large by a factor of the number of points in the transform. This is done to preserve accuracy during the transform as all data is stored as 16 bit integers. Thus the true result of the transform is given by:

true result = actual result * $2^{(scale + param)}$/ points transformed

where `param` is 0 for the DC and Nyquist points, and 1 elsewhere.

Example of use of the forward transform

Consider a simple example of the forward FFT in which we want to capture immediately, at a 50 kHz sampling rate, one sweep of 1024 samples of a waveform, window the sweep with a previously computed window waveform, compute the logarithmic power content over the (0 - 25 kHz) frequency range, and display the result on the host screen. The incoming signal should have been low pass filtered to reject frequencies above 25 kHz. The following command sequence would be sent to the 1401:

Example 13: Use of the forward transform

```
SS2,C,2048,2048,0                       Zero the window area
WRADR,2,2050,-16383                     Set amplitude for 1st harmonic
FFT,I,2048,2048;scale                   Produces a cosine wave, read scale
SS2,S,2048,2048,scale                   Shift data to correct size
RDADR,2,2048;upBy                       Find amplitude of first point
SS2,+,2048,2048,-upBy                   raise so window starts at 0
                    We now have a window at 2048

ADCMEM,F,2,0,2048,0,1,C,4,5     1024 samples of two byte data from channel 0
SM2,*,0,2048,2048,15                    Multiply by the pre-stored window function
FFT,FF,0,2048                           Real, followed by imaginary components
GAINPH,G                                Converted in place to log power and phase
SS2,B,0,1024;peak,where                 Find size and position of peak
                Display peak amplitude and position on screen
TOHOST,0,1024,buffer                    transfer gain part of data
```

Note that the array space of the ADC samples is re-used in turn for the result of the windowing multiplication, the `FFT` command and the `GAINPH` transformation. The only extra memory needed is for the window function.

The window has 1024 points (2048 bytes) with maximum value 32767. The scale factor of 2^{15} in SM2 normalises this to be nearly unity. There is no need to generate this window in the host and transmit it; the inverse FFT command can do the job for us in 1401 much faster.

GAINPH Log amplitude and Phase

The GAIN and PHase command converts the result of a FFT,FF,st,sz command into a more usable form for display. The data is changed from the sine and cosine amplitude form of the FFT output, to log amplitude and phase. The GAINPH command operates on the results of the last FFT command used. The general form is simply:

GAINPH,G

Consider the result of an 8 point FFT:

Point	0	1	2	3	4	5	6	7
Before GAINPH	R1	R2	R3	R4	RNy	I2	I3	I4
After GAINPH	Log1	Log2	Log3	Log4	Ph1	Ph2	Ph3	Ph4

Where Rn represents the n^{th} real value, In the n^{th} imaginary value, RNy is the real amplitude of the Nyquist frequency, LogN is a value that is proportional to the logarithm of the power and PhN a number proportional to the phase of the n^{th} component.

If scale is the scale factor returned by the FFT command, then LogN is given by:

$\texttt{LogN} = (1024/\log10(2)) * \log10((Rn^2+In^2) * 2^{2*(\text{scale + param})})$

where param is 0 for DC and Nyquist, 1 elsewhere.

$\texttt{PhN} = 32767 * \texttt{ATan}(\texttt{In}/\texttt{Rn}) / Pi$

where ATan(x) returns a result between -Pi and +Pi. It can be seen that a 3dB change in a signal level is given by a change of 1024 in the LogN values. The range of data values returnable for the LogN is from 0 to 32762, a range of 96 dB. The range of data values for the phase is from -32767 to plus 32767, representing -180 to +180 degrees (-Pi to +Pi).

0 corresponds to a sine wave of amplitude 1 bit. Thus:

result in dB = value(n) * 3.010/1024 with 0 dB = 1 bit

The GAINPH command has one argument field which must hold a G (for Go!). The argument is there to allow programs which perform automatic loading of commands to check for their presence by issuing just the command name which produces an error 254 if the command is there and 255 if not. All the information needed for the command is left in the 1401 by the FFT command. This means that the GAINPH command must follow a FFT command and the result array will overwrite the FFT result array.

If the GAINPH is executed without a previous FFT,FF,st,sz command, the result is undefined.

ADDPWR
Spectral averaging

This command is used together with `FFT` and `DLOGPWR` to perform spectral averaging. The command format is:

`ADDPWR,dest,reduce`

Where `dest` is the start of a double precision (32 bit per item) buffer used to accumulate the power derived from the last `FFT,FF` command. This buffer is of the same size as the `FFT,FF` data array, for if the FFT used a 512 byte array (being 128 real values and 128 imaginary, each of two byte significance) the `dest` array will be 128 4 byte double precision values.

The power is calculated in this command by summing the squares of the real and imaginary parts of the data from the FFT and multiplying this by:

$2^{2*(\text{scale factor returned by the FFT})-\text{reduce}}$

The result of this calculation is added into the appropriate position in the destination buffer. `ADDPWR` compensates internally for the differences in scaling between the DC and Nyquist components, and the rest of the spectrum.

The power is added to the array, to allow averaging, but in general the array may well not be zeroed when first used. To zero this double precision array, use `SS2,C,st,sz,0` remembering that the number of 16 bit elements in the array will be twice the number of 32 bit elements!

To ensure that the summing buffer will not overflow in use, the reduce parameter must be set such that:

2^{reduce} >= number of sweeps to be averaged

The reduce factor should be in the range 0 to 15.

If you start with a cosine wave of amplitude x, take its FFT and use `ADDPWR`, the increment to the corresponding bin in the destination array will be:

$x^2 * 2^{-\text{reduce}}$

DLOGPWR Log gain from ADDPWR

This command takes the result of a series of ADDPWR calls, and produces the average log power of the data. The command parameters are:

```
DLOGPWR,dest,source,sz,reduce,nsw
```

`dest` The result array starts at `dest`, and is of length `sz`/2 bytes. The format of the result array is the same as for the first half of the GAINPH result array. Thus the result data is 16 bits per data point.

`source` The double precision buffer used by the ADDPWR command, `sz` bytes long holding `sz/4` double precision powers.

`sz` The length of the `source` array in bytes, and twice the length of the `dest` array.

`reduce` The same number as was passed to the ADDPWR command. This will be used to allow for the shift down applied by ADDPWR. DLOGPWR increases the result by 1024 * `reduce`. (1024 is 3 dB here)

`nsw` The number of sweeps of data actually accumulated by ADDPWR.

The final result is given by:

$$dest(n) = (1024/\log_{10}(2)) * \log_{10}(source(n)/nsw) + 1024 * reduce$$

To save you calculating it: $1024/\log_{10}(2) = 3401.65$

Alternatively, to get the result in dB relative to 1 bit:

Result = `dest(n) * 3.010/1024` dB

A 1 Volt peak sine wave (0.7 Volt r.m.s.) on this scale is 76.3 dB.

Typical use of commands to average spectra

A typical use of these commands to produce an averaged spectrum is shown symbolically below. Lines in square brackets are transmitted to the 1401. Data is logged into locations 0 to 511 and is then multiplied by a data window that has been sent from the host to 1401, using the SM2 command. An FFT is taken and the result is accumulated with the ADDPWR command. This is repeated for the required number of sweeps. Finally the DLOGPWR command is used to produce the result from the summed data.

Example 14: Averaged spectra

```
                    Start by building a windows as for example 13
nsweeps = 10                                  Set number of sweeps to average
SS2,C,1024,1024,0                             zero the summing buffer
TO1401,4096,512,host address                  send the window to 1401
for i = 1 to nsweeps
  begin
  ADCMEM,F,2,0,512,0,1,C,10,10                get 512 bytes of data
  SM2,*,0,4096,512,15                         use window assumed at 4096
  FFT,FF,0,512                                take FFT of the data
  ADDPWR,1024,4                               sum into 1024 to 2047
  end
DLOGPWR,512,1024,512,4,nsweeps                take log of result
                    Find and display peak as in example 13
```

Event time processing

This group of commands is useful for acquiring and analysing data in the form of times of events, normally in response to a stimulus. Common users of this form of data are: psychologists, monitoring the responses of one or more subjects by means of push buttons, to stimuli which may be visual or auditory, and physiologists studying nerve firing patterns on one or more channels. Nerve signals must be amplified and brought to TTL levels by the internal 1401-18 optional 8 channel event conditioner, or by equipment external to the 1401.

The signal acquisition commands described below are all interrupt driven which means that the 1401 (and the host) are free to do other things at the same time, perhaps providing the stimulus pattern - such as controlling lights, to a programme which may be preset, or may be a function of the results. Note that it is not possible to run more than one of these event time collection commands at once.

The commands can handle quite fast data rates, of order 150 kHz for micro1401, 200 kHz for the Micro1401 and more than 250 kHz for the Power1401.

Single channel commands use event inputs 0 and 1; the digital inputs, bits 8 to 15 are used for the multi-channel commands. The user may choose to measure times on either the rising or the falling edges of the digital input signals. These inputs are selected as channels 0 to 7 with 0 corresponding to digital input 8 and 7 to digital input 15.

Trigger signals, if used, should be connected to event 1 and will be active on the falling edge unless this is altered by the EVENT command. The data capture commands are:

Command	page	Function
PSTH	70	Single channel Post Stimulus Time Histogram using event 1 as the stimulus and event 0 for the response.
PSTHM	72	Multi channel Post Stimulus Time Histogram using event 1 as stimulus and selected digital inputs as the responses.
INTH	73	Single channel Interval Histogram with event 0 as input and event 1 as an optional enable.
INTHM	74	Multi channel Interval Histogram: event 1 as (optional) enable and selected digital inputs for data.
AUDAT	75	Reads the absolute times of inputs on events 0 and 1 into two separate arrays. This data is then suitable for use as input to AUCR and AUINTH.
AUDATM	77	As AUDAT but uses the digital inputs in place of events 0 and 1. Up to 8 channels with optional enable on event 1.
AUCR	78	Routine to process the output from AUDAT and AUDATM. This will produce either a PSTH or cross correlation with each bin being a power of two clock ticks wide.
AUINTH	78	Processes an array of absolute times of events, giving an interval analysis. Time is compressible by powers of 2.

At the end of one of these commands, the data is stored in arrays in the 1401 memory. Data captured by the interrupt-driven commands can be transferred to the host for display during data capture.

Before running any of the data collection commands, users will make an estimate of the time of interest after each stimulus. This will be the sweep time of the analysis. The number of bins in the sweep, and the timing resolution must be considered at this stage. The number of clock ticks per bin can only be varied by factors of two so if fine adjustment is needed, it is done with the number of time units per clock tick.

Standardised argument names

In the following descriptions of the various commands, many similar arguments occur. These have been given standardised names, as below, the details of which are not repeated in the specific descriptions.

`st` The start of an array in the user storage space in the 1401. The array must be at an even address.

`sz` The size of an array in bytes in the 1401. Event time data is stored as 16 bit unsigned integers, so `sz` must always be even! Odd values of `sz` cause an error 253. For the histogram commands, `sz/2` is the number of bins.

`n` A channel number in the range 0 to 7. Channel 0 refers to digital input bit 8, channel 1 to bit 9, channel 2 to bit 10 and so on up to channel 7 and bit 15.

`[x]` Single items in square brackets are optional.

`unit` Where a time unit is indicated, the commands expect '`M`' for milliseconds or '`U`' for microseconds.

`time` The number of time units per bin or per clock tick.

`edge` If present, defines the active edge of the signal into the digital inputs. '-' sets falling edge, '+' sets rising edge and if absent, the default is falling edge.

`[T]` Optional enable on event 1 (E1) front panel input. If included, no data will be taken nor will the clock start until there is a falling edge into event 1.

`status` A value returned by a command, currently from this range:

0	Command never started or terminated
1	Awaiting the trigger on event 1
2	Command logging data

`done` Number of sweeps or events when the query command is received.

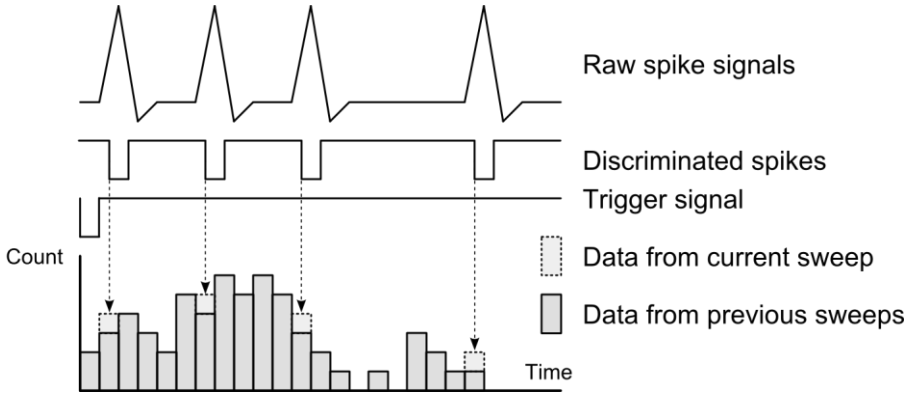
`ovflws` A value returned by a command; a count of the number of times any of the bins in a `PSTH` or `INTH` has overflowed. To overflow, the bin has to hold 65535 and have an extra count added. The bin then appears to hold 0, but should hold 65536. Parameter `ovflws` increments each time any overflow occurs. The counter is an 8-bit number for standard 1401.

`pwof2` Used in the processing commands to fit a power of two counts into a bin rather than just 1. This is useful if your data was sampled at a resolution of 1 ms but you want to display a histogram of a second in width with only 128 bins. You set `pwof2` to 3, i.e. divide original time by 23(=8) so that 128 bins will last 1024 milliseconds, each bin 8 ms wide. Single channel post stimulus time histograms (`PSTH`)

PSTH Post stimulus time histogram

The PSTH command allows the simple collection of Post Stimulus Time Histogram data based on TTL compatible data pulses on the 1401 E0 and E1 front panel inputs. The event 1 input is the 'stimulus' and the event 0 input the 'responses'. A multi-channel version, PSTHM, is described below.

Raw and discriminated signals and PSTH



This is the basic principle of the post stimulus time histogram. A raw data signal, if not already TTL or switch closure, is discriminated by hardware to produce a series of 'event' pulses. A 'stimulus' or 'trigger' signal is also discriminated to produce a pulse. This 'stimulus' pulse starts a clock that records the times after the stimulus at which the event pulses occur. Each event causes a histogram bin to be incremented, the selected bin corresponding to the time of the event after the stimulus.

This process can be repeated many times, the final histogram being a representation of the probability of an event occurring at a given time after the stimulus. The variants of the command are:

`PSTH,G,st,sz,time,unit,sweeps`	Set up and go
`PSTH,?;status,done,ovflws`	Report state of command
`PSTH,P;offs`	Report offset into sweep
`PSTH,S`	Stop after next sweep
`PSTH,K`	Kill PSTH command

Set up and go

`PSTH,G,st,sz,time,unit,sweeps` clears any PSTH command in progress and arms a new one, leaving the 1401 waiting for a stimulus on event 1.

Report state

`PSTH,?;status,done,ovflws` reports on the current state of the PSTH command, returning three numbers. If the PSTH has finished or has never been started, `status` is returned 0 and done is returned as the number of complete sweeps recorded when the `PSTH,?` command was received, and `ovflws` is the number of times a bin has overflowed, i.e. the number of times more than 65535 events have been stored in any bin.

Report offset

`PSTH,P` returns 0 if a sweep is not in progress, otherwise it returns the current byte offset in the sweep. This allows monitoring of the sweep state for slow sweeps.

Stop the command

`PSTH,S` stops the data recording at the next end of sweep. It does not cause another sweep to be recorded if the previous `PSTH,G...` command has already finished.

`PSTH,K` kills the PSTH command immediately, even if currently running.

Limitations on bin widths

The width of each data bin can be set in either milliseconds, or in microseconds. The minimum time per bin is 2 microseconds, each bin can be no more than 65.535 seconds wide. There is a relationship between the time per bin and the maximum number of bins which is:

number of bins * time per bin <= 71.5 minutes

Accuracy of bin widths

Any bin width below 66 milliseconds will be set exactly. A bin width request of more than 66 milliseconds is subject to an error of less than 1 part in 32767. These errors will always make the bins too short. This is because the period is a number in the range 2 to 65535 times a power of two (in microseconds). Any period that can be exactly represented in this way will be; 1000 ms is represented exactly, but 1001 ms is represented as 1000.992 milliseconds.

Sweep time-out

We do not want sweeps to continue forever! A sweep is terminated by: another stimulus, or when the time exceeds `unit*time*sz/2` the sweep terminates at the first of: another event, or the automatic time-out. The time-out periods are set by:

Possible time-outs(microseconds) = 65536 * n (n is 2 to 65536)

The minimum time-out is thus 131 milliseconds, and the increments in the time-out are in 65.5 millisecond steps. This implies a limit to the spacing of stimulus pulses to the `PSTH` command.

Example of use of the PSTH command

This example allows the user to set up a `PSTH` and to kill it by pressing the `K` key on the terminal or to stop it after one more sweep by pressing any other key. A report of the number of sweeps done is maintained. The example could easily be extended to display the `PSTH` as it builds up using the `D` command.

Example 15: Use of the PSTH command

Set up communication with the 1401 and load the command `PSTH`
Ask the user to supply `binwidth` and sweep `number` information at the keyboard.
Clear the 1401 state and zero a data array by sending:

```
CLEAR;SS2,C,0,size,0
```

Set up and start the PSTH acquisition by sending:

```
PSTH,G,0,size,binwidth,U,number
```

Read the state of the acquisition by reading in the data returned by the 1401 in response to sending:

```
PSTH,?
```

Display the number of sweeps done on the host screen, checking if the user has requested termination, or else that the full number of sweeps is done anyway. If termination is requested, kill the command by sending:

```
PSTH,K
```

PSTHM multi-channel PSTH

The PSTHM command provides interrupt-driven collection and sorting into histogram bins of up to 8 trains of TTL compatible data pulses, received on the digital input connector, with stimulus pulses entered on the event 1 (E1) input.

Command variants

`PSTHM,An,st,sz`	Set up call for channel n (0-7)
`PSTHM,G,rpts,time,unit[,edge]`	The Go call to start the PSTH
`PSTHM,?;status,done,ovflws`	Report state and overflows
`PSTHM,S`	Stop after the next sweep
`PSTHM,K`	Kills the command

Set up call for each channel

`PSTHM,An,st,sz` must be used first, to set up for each input to be used, the data array start `st` and number of bins `sz/2`. Inputs may be set up in any order, and any combination of the 8 may be used. Low order inputs are processed first so slightly faster performance is available if, when only a few channels are needed, low numbered channels only are used.

Input connections for multi-channel timing

Event channel number	0	1	2	3	4	5	6	7	Gnd
Digital input pin	17	4	16	3	15	2	14	1	13

You can use the front panel event inputs for channels 0 and 1 and can route the stimulus pulse via the Trigger input using:

```
EVENT,T,2;EVENT,E,192;
```

The restrictions on the bin widths and the number of bins are explained in the description of the PSTH command. The same restrictions apply to PSTHM. A sweep is over when:

- A response or a stimulus occurs outside the range of the histogram.
- The sweep is timed out. The time out is in units of 65.536 ms, with a minimum time-out of two units (131.072 ms).

Go, start logging

`PSTHM,G,rpts,time,unit[,edge]` starts the logging, with `rpts` as the number of sweeps, with `time` and `unit` setting the bin width. `edge` sets the active edge of the inputs.

Report the state of the command

`PSTHM,?;status,done,ovflws` returns three numbers to the host computer which describe the current state of the command (see page 69 for a full description).

Stop the command

`PSTHM,S` stops data capture at the next end of sweep. This provides a method of terminating the command early, before `rpts` sweeps have been completed. The `PSTHM,G...` command may be used to restart sampling after the stop call without the need for the `PSTHM,A...` command to be re-used.

INTH
Single channel interval histogram

The INTH command generates real time interval histograms of pre-discriminated signals fed as TTL pulses into the E0/Event 0 input. Data capture waits for the first pulse on E1/Event 1 if the optional T is used. The command is entirely interrupt driven and while running may be interrogated to determine the current state of the sampling. There is a multi-channel command INTHM, described below.

The INTH command is very similar in operation to the PSTH command, but the intervals between pulses on the E0/Event 0 input determine the bin to be incremented rather than the interval between the sweep start and the pulse.

Command variants

`INTH,G[T],st,sz,time,unit,count`	Set up and go
`INTH,?;status,done,ovflws`	Query state of command
`INTH,S`	Set one more interval
`INTH,K`	Kill the command

The arguments are standard, as described on page 69, except for `count` which is the number of intervals to be logged. Intervals which fall outside the histogram are included in the count.

Set up and go

`INTH,G[T],st,sz,time,unit,count` sets up the 1401 either to log event intervals immediately (without the T) or to wait for an event on the E1 front panel input if T is used (as in PSTH above) before starting to log intervals. The first event logged on E0 does not cause an entry in the histogram as no interval between events is defined. If 1000 intervals are requested, 1001 events are needed.

Report state of the command

`INTH,?;status,done,ovflws` returns the usual meanings of `status` and `ovflws`; done in this case is the total number of events, whether they contributed a point to the histogram or not.

Stop the command

`INTH,S` stops the command after the next interval, while `INTH,K` kills it immediately, as usual.

Example of use of the INTH command

This example lets the user set up an INTH and to kill it by pressing the K key on the terminal, or to stop it after one more interval by pressing any other key. A report is kept of the number of intervals logged. The example could easily be extended to display the INTH as it builds up using the D command.

Example 16: Use of the INTH command

```
Establish communication with the 1401
Load the INTH command from disk, if necessary
Ask for parameters from the keyboard
        CLEAR;SS2,C,0,size,0                    Clear 1401 and zero the array
        INTH,G,0,size,binwidth,U,count          Start INTH running
        INTH,?                  Check state of sampling by reading responses
Check the first number to see if all sweeps are done; if not, check the host keyboard for
stop requests. Send, as appropriate:
        INTH,S                                  if any key but K was pressed, or
        INTH,K                                  for an immediate Kill
```

INTHM multi-channel interval histograms

The INTHM command provides interrupt-driven collection and sorting into histogram bins of the intervals between events on up to 8 trains of TTL compatible data pulses, received on the digital input connector, with an optional start pulse expected on the front panel Event 1 or rear E1 input, see the EVENT command.

Command variants

`INTHM,An,st,sz`	Array set up command
`INTHM,G[T],tmo,time,unit[,edge]`	Go command, as in PSTHM
`INTHM,?;status,tmo,ovflws`	State/overflow query
`INTHM,S;tmo`	Stop/progress command
`INTHM,K`	Kill the command

The arguments are all as standard (described on page 69) except for `tmo` (described below) and the input connector configuration is as shown for `PSTHM` on page 72.

General use of the `INTHM` command starts with a call to `INTHM,K` to ensure that nothing is left from the previous use of the command. This is followed by calls to the `INTHM,An...` command to set the bins in the histogram for each channel. Remember that the `sz` must be set to twice the number of bins as `sz` is a byte parameter and the bins are two bytes long each. Once a channel is set, it remains set until `INTHM,K` is run.

Set up and go

`INTHM,G[T],tmo,time,unit[,edge]` starts data capture and also sets the time increment and mode of starting. If the start of the data capture must coincide with an external event, triggered starting is selected by using the optional `T`; data will otherwise be captured from the moment the `INTHM,G...` command is issued. Data acquisition will stop after `tmo` units of 65.536 milliseconds each, in the range 2 to 65536, that is from 131 milliseconds to over an hour. The active edge of the input signals is set by edge.

Report state of the command

`INTHM,?;status,tmo,ovflws` returns the standard `status` and `ovflws` information about the `INTHM` command as it runs, but `tmo` is also returned as a number between 0 and 65535 representing the proportion of the time out elapsed. So if it were returned as 32767, half of the time out would have elapsed. `ovflws` is 0 if no overflow happened and is non-zero if there were overflows.

Stop the command

`INTHM,S;tmo` stops data acquisition, and returns `tmo` as the proportion of the elapsed time out as in `INTHM,?...` above. Once stopped, the `INTHM,G...` command may be used to restart the data acquisition.

`INTHM,K` completely kills the `INTHM` command. This is the only way to make the command ‘forget’ about previously selected inputs.

Note that the first event on each channel is used to generate the period to the second event, the first does not cause any data to be recorded.

AUDAT Absolute event time capture

Some forms of event time analysis, such as correlations, need the times of all the events. The AUDAT family of commands is provided for these situations. The family has four sections: acquisition and processing, for single/dual and multi-channel use. AUDMR is the mass RAM version described in Appendix A.

The dual channel data acquisition command is AUDAT, which has a speed advantage over the AUDATM command below. The maximum rates per channel (sampling 5,000 points, both channels at the same rate) are:

AUDAT maximum rates in kHz

Channels	micro1401	Micro1401	Power1401
1	150 kHz	200 kHz	>250 kHz
2	95 kHz	127 kHz	>250 kHz

If you exceed these limits, data points may be lost or timings may become inaccurate. The command stops when the data arrays are full, or when the time implied by cycles, or the rpts number of sweeps is reached, or when the Stop or Kill commands are used.

For very large (or unpredictable) amounts of data, AUDAT can be used in a circular mode with half buffer flags (like ADCMEM,I) so that completed half buffers can be written to a disk file.

The data gathered by AUDAT or AUDATM is usually processed by the AUCR and AUINTH commands below. Data from AUDMR, and AUDAT in circular mode is generally copied to the host disk for processing by more specialised code.

Command variants

`AUDAT,G[T],e0st,e0sz,e1st,e1sz,time,unit[,cycles]`	Normal setup
`AUDAT,C[T],e0st,e0sz,e1st,e1sz,time,unit[,rpts]`	Circular setup
`AUDAT,?;status,e0byt,e1byt`	Query normal mode status
`AUDAT,?;status,e0byt,e1byt,e0hbufs,e1hbufs`	Query circular status
`AUDAT,S;e0byt,e1byt`	Stop normal mode command
`AUDAT,S;e0byt,e1byt,e0hbufs,e1hbufs`	Stop circular mode command
`AUDAT,K`	Kill the command

Normal mode set up and go

AUDAT,G... is used to set up and start the capture of absolute event times from E0 and E1 front panel inputs in non-circular mode. e0st and e0sz define the start and size of the area for storage of the event 0 times, and the event 1 area is defined by e1st and e1sz. The command may be run on event 0 only by specifying zero start and size of the event 1 array. It is not possible to use just event 1.

The event times are stored as 16 bit (2 byte) numbers. To allow us to save the times of events past 65535 clock ticks we insert an extra marker of value 32768 into the data array each time the clock reaches 32768 and restart the clock from 0. These extra values never denote an event. If an event happens to fall on a multiple of 32768 clock ticks it is given the time 0.

This is made clearer by an example. To save data whose event times are 1234, 28387, 32768, 33000, 59873, 68281, 254000 we would save the values 1234, 28387, 32768, 0, 232, 27105, 32768, 2745, 32768, 32768, 32768, 32768, 32768, 24624.

The analysis commands AUCR and AUINTH sort out these extra 32768s, so most users need never be aware of them. Only users who transfer the arrays of times to the host will need to be aware of these overflow markers.

If the optional T is used, the command will wait for the first falling edge on the event 1 input before starting the clocks.

The clock tick period is (`time * unit`) and must be in the range 2 microseconds to 65535 microseconds (1 to 65535 with the Micro2/3 and Power2/3). The unit is specified as `M` for milliseconds, `U` for microseconds, or `N` for nanoseconds (not micro1401). If unit is milliseconds, the maximum value of time is 65. If the units are nanoseconds, the value must be an exact multiple of 100.

`cycles` is an optional argument in the `AUDAT` and `AUDATM` commands, defaulting to 1. It sets the maximum time that the command runs for, which is given by:

```
cycles * unit * time * 32768
```

so with millisecond units, time = 1, cycles = 10, the maximum time would be 327.68 seconds (10 * ms * 1 * 32768).

Circular mode set up and go

`AUDAT,C...` is used to setup the capture of event times in circular mode and is identical to `AUDAT,G...` except that:

- the `sz` argument must be divisible by 4, as the array will be split into two equal parts, both to be filled with 2 byte data, and
- the final `rpts` argument which replaces `cycles` sets not the maximum time for the command but the number of complete buffers at which to stop each channel.

Report the state of the command

`AUDAT,?;...` returns the current state of the `AUDAT` command as usual, and the number of bytes of the event 0 and event 1 arrays that have been filled. Remember that there are two bytes per data point! If it is called after a circular mode `AUDAT,C...` command, the two extra arguments `e0hbufs` and `e1hbufs` show the number of complete half buffers that have been filled on each channel.

Stopping the command

`AUDAT,S;e0byt,e1byt,(e0hbufs,e1hbufs)` stops the sampling and returns the number of bytes of event 0 (`e0byt`) and event 1(`e1byt`) data gathered so far. In circular mode only, the number of half buffers filled is also returned (`e0hbufs` and `e1hbufs`). `AUDAT,K` kills the command, as usual.

AUDATM multi-channel event time capture

For more than two channels, the AUDATM command is used to get the times for subsequent processing. Digital inputs 8 to 15 are used in place of the events 0 and 1, the input connector configuration is as shown for PSTHM on page 72. The sub-commands are very similar to those in AUDAT, with extensions to handle the increased numbers of channels. AUDATM does not have a circular mode.

AUDATM maximum rates in kHz

Channels	micro1401	Micro1401	Power1401
1	87 kHz	200 kHz	>200 kHz
2	71 kHz	200 kHz	>200 kHz
4	52 kHz	200 kHz	>200 kHz
8	34 kHz	164 kHz	>200 kHz

The table shows the maximum rates in kHz for a number of channels fed the same data. The test sampled 2,500 events per channel. Faster rates can drop points with no warning.

Command variants

`AUDATM,An,st,sz`	Set up for channel n (0-7)
`AUDATM,G[T],0,time,unit,edge[,cycles]`	Go and start clock
`AUDATM,?n;status,chstat,done`	Query channel n status
`AUDATM,?;status`	Query command status
`AUDATM,Sn;done`	Stop channel n only
`AUDATM,S`	Stop all channels
`AUDATM,K`	Kill the command

Set up a channel

`AUDATM,An,st,sz` sets up the data area for each channel to be used. The absolute times of the events on each channel will be stored from `st` with space for `sz/2` events and clock overflow markers. No check is made whether the data area of each channel is free of the others. `cycles` is optional, as in AUDAT above.

Go, start the command

`AUDATM,G...` starts up the command and sets the clock rate. The active edge of the input signals may be selected with the value of `edge`. `T`, if used, causes the command to wait for a falling edge on the event 1 (E1) input before starting. The command is somewhat unusual as the 0 is a dummy field that is to allow for future expansion.

The same restrictions as in the AUDAT command apply to the `time` and `unit` fields.

Report channel n status

`AUDATM,?n;status,chstat,done` returns the overall command `status` and the status of channel `n` as `chstat` which is 0 if the buffer has filled (or sampling has not started) and 2 while the buffer is filling. `done` is the number of bytes recorded for channel `n`.

Overall command status

`AUDATM,?;status` returns the overall command status, which is faster to transmit than the fuller information above.

Stop one channel

`AUDATM,Sn;done` stops channel `n` and returns the number of event times and overflow markers collected on that channel as `done`. The Standard 1401 is different and returns twice this number (i.e. the number of collected bytes).

Stop all channels

`AUDATM,S` stops all channels and as usual, `AUDATM,K` kills the command. The command stops when either all the arrays are full, or when the Stop or Kill commands are used.

AUCR and AUINTH commands

These commands process the completed results from AUDAT or AUDATM and produce two types of PSTH or a Cross correlation, and AUINTH generates an interval histogram. The response time region can be offset backwards or forwards from the stimulus time to produce peri-event histograms.

Command variants

```
AUCR,P,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws      Standard
AUCR,C,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws      Curtailed
AUCR,X,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws      Cross
AUINTH,G,st,sz,dst,dsz,pwof2;ovflws                           Interval histogram
```

AUCR processes two data arrays typically from AUDAT and AUDATM to produce a third data array. sst and ssz define the start and size of the 'stimulus' time arrays. rst and rsz define the 'response' time arrays and dst, dsz define the result array. pwof2 sets the clock ticks per result bin (see page 77). nsweeps is the sweeps of data processed. The array lengths are in bytes, so the number of times (for the stimulus and response arrays) and the number of bins (for the result array) are half the byte size of each array.

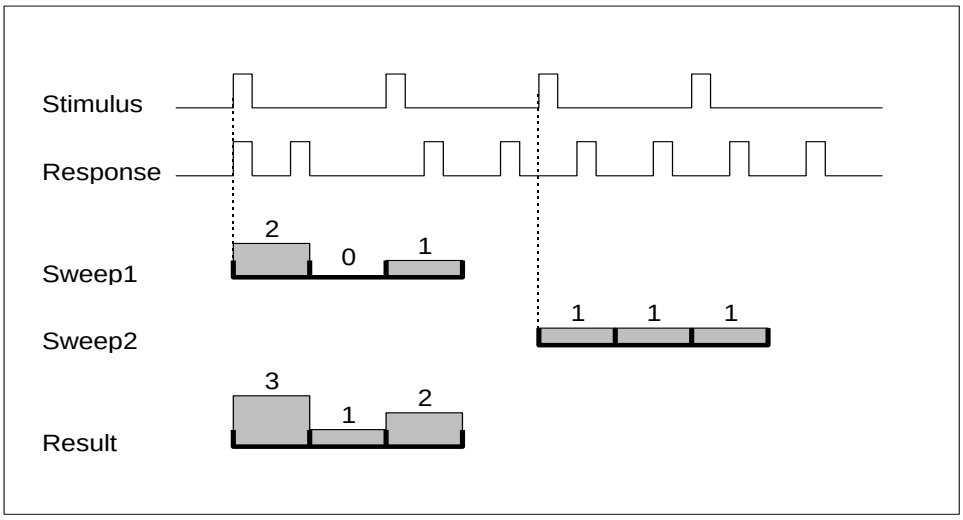
The optional offs (range ±32767, default 0) offsets the histogram start time in clock ticks relative to the stimulus. Do not make the offset more than the inter-stimulus time or no events will be recorded! Commands terminate when an attempt is made to access a data element beyond the end of either data input array. This avoids the apparent drop at the end of the result histogram due to the last sweeps having less data to work with.

In the diagrammatic examples, the stimulus array is 4 data points long, the response array is 8 data points long, and the result array is 3 bins long.

Make a standard PSTH

AUCR,P,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws is intended to produce the same result as if the original data had been fed into the PSTH command. In the example the first stimulus starts a sweep of collection. The second stimulus is ignored, because the first sweep of 3 time units has not been completed. The fourth response is ignored because the second sweep has not yet been started by the third stimulus. nsweeps is 2.

How the PSTH is built

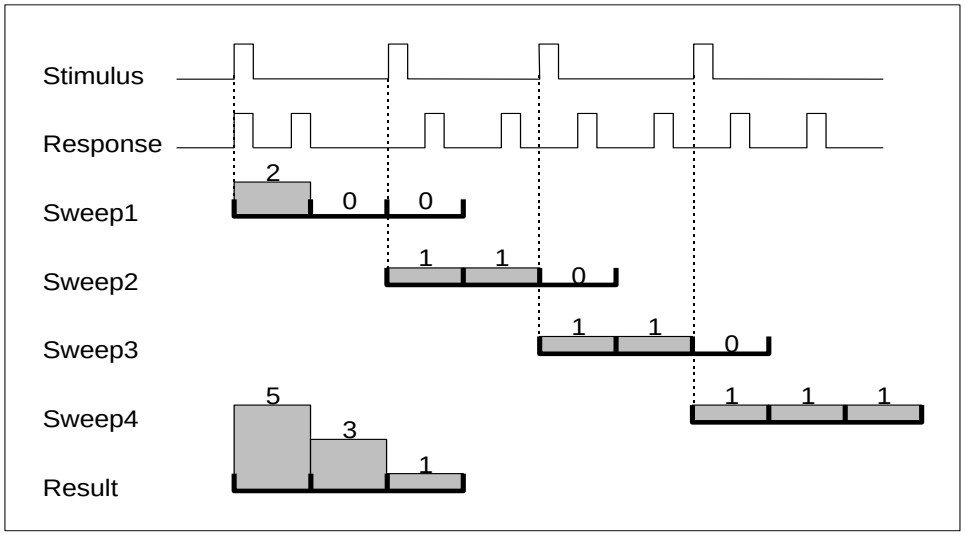


Make a curtailed PSTH

AUCR,C,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws makes a post stimulus time histogram from the data, but sweeps are terminated either by being timed out as in AUCR,P or by the next stimulus, thus producing a different result with the same data. This would be the type chosen if activity died away rapidly and one response would

not be expected to last into the next stimulus. Care should be taken with the value of `offs`, if used. If the inter-stimulus time is less than the offset, no points will be returned.

How the curtailed PSTH is built

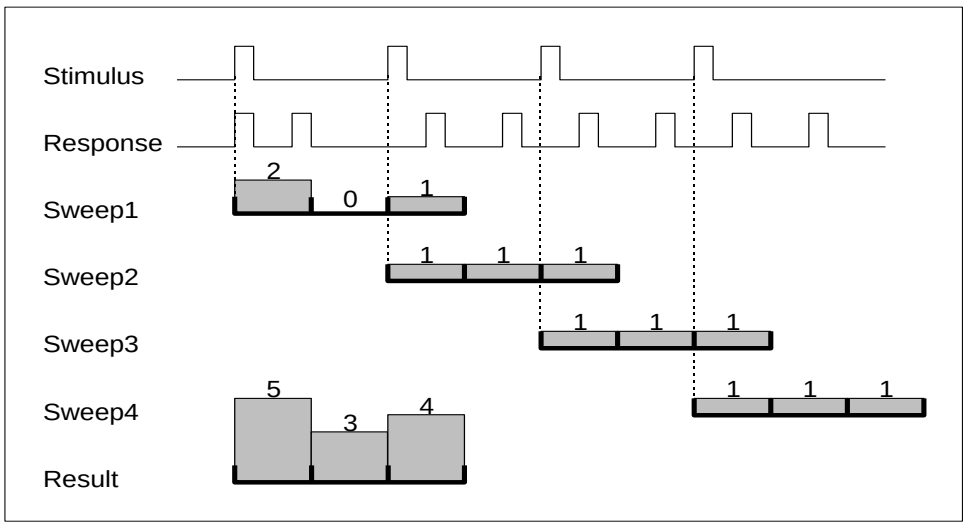


In this example, the first 3 sweeps are curtailed by new stimuli, so there is nothing in the third bins until the fourth and last sweep, which is not curtailed. `nsweeps` = 4.

Cross correlation

`AUCR,X,sst,ssz,rst,rsz,dst,dsz,pwof2[,offs];nswps,ovflws` returns a result showing the probability distribution of the responses occurring after each stimulus, relative to the stimulus. It is similar to the `AUCR,P...` command except that more than one sweep may be in operation at any one time.

How the cross correlation is built



Each stimulus is used as the start of a sweep, so if two stimuli occur within one sweep time of each other, the second stimulus will generate a new sweep. Compare this with the `AUCR,P...` command where the second stimulus would be ignored. `nsweeps` = 4.

Make an interval histogram

`The AUINTH,G,st,sz,dst,dsz,pwof2;ovflws` command creates an interval histogram array from a data array of increasing times such as those produced by the `AUDAT` or `AUDATM` commands. The parameters have the same meanings as in `AUCR`.

There are situations when the host computer for 1401 can only give slow responses, especially if a slow interface, for example serial line control, is used. There are also applications where users wish to run 1401 without an active connection to a host computer. RUNCMD and VAR address these situations.

RUNCMD Run from an internal list of commands

RUNCMD is a loadable command that programs the 1401 to run a sequence of commands, with simple branching control, and with no involvement by the host computer. This is useful where the 1401 is run on a slow link to the host, such as a serial line. This facility can also get more work from the host. Instead of the host waiting for operations in the 1401, the 1401 can be handed the sequence of activities and left to get on with it.

The commands RUNCMD and VAR can also form the basis of a stand alone 1401 running completely free of a host. With standard 1401, command sequences and initial values of variables are stored in ROM and copied into RAM during power up. The 1401 can then perform set tasks under the control of data read by itself, such as digital or event inputs.

Redirection of output

When the 1401 runs an internal sequence, output that would normally be sent to the host is intercepted and goes to local variables within the 1401. The variables are available for arithmetic or logical operations and to control the flow of execution of the commands.

The first number on each output line from the 1401 to the host is placed in local variable A, the second in local variable B and so on. Local variables which are required to hold permanent data should thus be allocated away from the start of the alphabet! Character strings which cannot be interpreted as integers are ignored. CLIST is the only standard command that produces output which would be ignored.

You can send output to named local variables by ending a command with a colon and a local variable. The following reads ADC channels 0 and 1 into local variables S and T:

```
ADC,0 1:S
```

You can also force output back to the host, even when a sequence is running by using a question mark in place of the local variable. The following command sends the output of the same ADC command back to the host, and not to the local variables A and B:

```
ADC,0 1:?
```

You can use these output redirection facilities at any time; they are described here because they are most useful with RUNCMD.

RUNCMD variants

The first group of RUNCMD variants is:

`RUNCMD,L`	Store the following sequence of commands
`END`	Terminates the command sequence (MUST be upper case)
`RUNCMD,G`	Take input/output from internal buffers (run sequence)
`RUNCMD,D`	Take input/output from host (end sequence)
`RUNCMD,R`	Run the sequence after a hardware reset (standard 1401 only)

Branching variants

`RUNCMD,BR,com`	Branch unconditionally to command number `com`
`RUNCMD,BP,com,arg1,arg2`	Branch if `arg2>=arg1` (signed)
`RUNCMD,BM,com,arg1,arg2`	Branch if `arg2<arg1` (signed)
`RUNCMD,BG,com,arg1,arg2`	Branch if `arg2>=arg1` (unsigned)
`RUNCMD,BL,com,arg1,arg2`	Branch if `arg2<arg1` (unsigned)
`RUNCMD,BE,com,arg1,arg2`	Branch if `arg2=arg1`
`RUNCMD,BN,com,arg1,arg2`	Branch if `arg2<>arg1`

Loading the input buffer; the L command

Space for the sequence is reserved when the command is loaded. The maximum sequence size is 2000 characters (4000 for Power1401). A sequence is loaded by:

`RUNCMD,L`	Flag the start of a sequence
`COMMAND1`	A command to be part of the sequence
`...`	More commands
`END`	Marks the end of the sequence (must be upper case)

The following lines sent to the 1401 would load a sequence that reads in the DC value of an array and removes it:

`RUNCMD,L`	Flag following text is sequence
`SS2,A,0,2000`	Get mean value of area into local variable A
`SS2,+,0,2000,-A`	Add minus the mean level to the data
`END`	End of the sequence (must be upper case)

To run the sequence, use the `RUNCMD,G` command. Individual commands in a sequence need not be on separate lines as long as they are separated by a semicolon.

Running and stopping a sequence

The `RUNCMD,G` command starts the sequence running. The sequence stops when it reaches the end, or a `RUNCMD,D`.

Any characters sent to the 1401 while it is running a sequence are stored in the 1401 communications buffers and will be processed when the sequence stops. If the communications buffer becomes full, the host must wait until communication is restored by `END` or `RUNCMD,D`. When the 1401 reset line is pulsed, all the 1401 hardware is reset, but any sequence loaded by the `RUNCMD,L` command is preserved in memory.

The Branch commands

These commands are valid when running from the internal buffer. It is an error to use them at any other time. In all these commands the second argument is the number of a command in the internal buffer to which control should be passed if the condition set in the branch is met. Commands are numbered 1 to N where there are N commands in the buffer. If a command number is given that is not in the buffer the sequence terminates.

The `BR` command jumps to the specified command in the buffer by number. The remaining commands (`BP`, `BM`, `BG`, `BL`, `BE`, `BN`) branch to the specified command if the last two arguments meet the appropriate condition. These arguments may be numbers or single letters corresponding to local variables. If the condition is not met the branch is not taken and the next command is executed.

Example sequence

This example shows a more ambitious application of `RUNCMD` in which a larger sequence is sent as text. The program in the host should load the sequence into the 1401, and when the sequence is initiated, by sending `RUNCMD,G` the link to the host may be disconnected.

The 1401 is told to respond to digital inputs 0 to 3 thus:

Digital input 0 set-	Sample and display 1024 points at 1 kHz
Digital input 1 set-	Window the display
Digital input 2 set-	Perform an FFT
Digital input 3 set-	Produce a power spectrum

This sequence would be loaded into the 1401 by sending the following commands (neither the command number in the sequence nor the comment is sent):

`RUNCMD,L`	set up to load
`SS2,C,4096,2048,0`	send first command of the sequence
	... the rest of the commands ...

```
RUNCMD,BR,6                                    the last command in the sequence
END                                            terminate the sequence
```

Once the whole text has been sent, the sequence is started by sending the `RUNCMD,G` command. There is no way out of this sequence!

No.	Command sent to 1401	Comment (not sent!)
1	`SS2,C,4096,2048,0`	zero an array for the spectrum
2	`WRADR,2,4098,-16383`	set a solitary cosine component
3	`FFT,I,4096,2048`	Generate a cosine array
4	`SS2,S,4096,2048,1`	Scale to make the window
5	`SS2,+,4096,2048,16384`	DC shift the window
6	`DIG,I,0`	Main loop; read digital input 0
7	`RUNCMD,BN,11,1,A`	go on if bit 0 not set
8	`D,0`	kill display
9	`ADCMEM,F,2,0,2048,0,1,C,10,100`	Sample
10	`D,Y2,64,0,2048`	and display
11	`DIG,I,1`	read digital input 1
12	`RUNCMD,BN,16,1,A`	go on if not set
13	`D,0`	kill display
14	`SM2,*,0,4096,2048,15`	window the data
15	`D,Y2,64,0,2048`	display
16	`DIG,I,2`	read digital input 2
17	`RUNCMD,BN,22,1,A`	go on if not set
18	`D,0`	kill display
19	`SM2,C,2048,0,2048`	make a copy of data
20	`FFT,FF,2048,2048`	do FFT
21	`D,Y2,64,2048,2048`	display
22	`DIG,I,3`	sample bit 3
23	`RUNCMD,BN,29,1,A`	go on if not set
24	`D,0`	kill display
25	`SM2,C,2048,0,2048`	make a copy
26	`FFT,FF,2048,2048`	do FFT
27	`GAINPH,G`	and gain phase
28	`D,Y2,128,2048,1024`	display power
29	`RUNCMD,BR,6`	branch back to 6

RUNCMD speed

Sequences of commands usually run faster when loaded with `RUNCMD` rather than controlled via the host because the overhead time between commands is reduced. The following loop, which writes a descending ramp to the digital output port took 1 seconds on a micro1401, 0.08 seconds in a Micro1401 and about 0.03 seconds on a Power1401, but the equivalent loop, run from the host (an IBM PC), took 8.6 seconds:

```
DIG,S,255                                      set the bi-directional bits to output
VAR,S,X,1000                                   set variable X to 1000 (VAR is described overleaf)
DIG,O,X                                        write X to the output
VAR,D,X                                        decrement X
RUNCMD,BN,3,0,X                                branch back if X<>0
```

If the sequence involves reading data back, then the time gains are larger. The sequence below reads ADC channel 0 one thousand times.

```
VAR,S,X,1000                                   set variable X to 1000
ADC,0                                          read ADC 0 into variable A
VAR,D,X                                        decrement X
RUNCMD,BN,2,0,X                                branch back if X not 0
```

With a micro1401 it took 1 second, on a Micro1401 it took 0.13 seconds and about 0.04 seconds on a Power1401, compared with 12.8 seconds when run from the host. The real speed gains occur where the host/1401 interface is slow (e.g. RS232), or where the host can usefully be doing other work. For example, the user wants to capture 10 voltage signals, each triggered by E4. These triggers will be widely separated in time and the user doesn't want to tie up the host waiting for each trigger:

```
RUNCMD,L                          load the following sequence
VAR,S,X,10                        set 10 sweeps
VAR,S,S,0                         set start memory position to 0 (S=0)
VAR,S,Z,1000                      set size of data to 1000 bytes (Z=1000)
ADCMEM,F,2,S,Z,0,1,CT,10,3        capture ADC data to area starting at S
VAR,+,S,Z                         move start on by the size
VAR,D,X                           decrement X
RUNCMD,BN,4,0,X                   round again if X<>0
END                               flag the end of the sequence
```

VAR Manipulating local variables

The 1401 `VAR` command is used to set, increment and decrement local variables. The `VAR` command has these variants:

```
VAR,S,lvar,arg                    Set a local variable (lvar) to arg
VAR,+,lvar,arg                    Add arg to the lvar
VAR,I,lvar                        Add 1 to the lvar
VAR,D,lvar                        Subtract 1 from the lvar
VAR,?,lvar;value                  Return current lvar value
```

`lvar` The local variable to use (`A-Z`)

`arg` A numeric value, or another local variable!

`value` The current value of `lvar`

The following are examples of the use of the `VAR` command:

```
VAR,S,A,1                         Set variable A to 1 (A=1)
VAR,S,P,-2000                     Set P to -2000 (P=-2000)
VAR,S,Q,$1000                     Set Q to hexadecimal 1000 (Q=4096)
VAR,S,D,-E                        Set D to minus the value in E (D=-E)
VAR,+,X,-Y                        subtract Y from X (X=X-Y)
VAR,I,F                           Increment the value in F (F=F+1)
VAR,D,G                           Decrement the value in G (G=G-1)
VAR,?,A                           Return the value of A to the host
```

CONFIG Configuring the EEPROM memory

The micro1401, and Power1401 use an EEPROM (Electrically Erasable Programmable Read Only Memory) to store system information. We usually fit one with 128 bytes of storage but larger sizes (256, 512 or 1024) are possible. The Micro2/3 and Power2/3 emulate the EEPROM with flash memory. The data within the EEPROM is divided into a read-only private header and a read/write public data area. If the private header becomes corrupted contact CED for instructions. The access commands are:

Command variants

`CONFIG,RC,hoOff,max,code`	Read public area code
`CONFIG,RN,hoOff,max,n`	Read public area packet `n`
`CONFIG,WB,hoOff`	Write public area
`CONFIG,RH,hoOff,max`	Read private header
`CONFIG,WH,hoOff`	Write private header
`CONFIG,T,dev;type,rev,serial,tags,cpld`	Read top box information
`CONFIG,TV,dev,offs;val`	Read top box EEPROM data byte
`CONFIG,RT,hoOff,dev,offs,n`	Transfer top box EEPROM data to the host
`CONFIG,WT,hoOff,dev,offs,n`	Write top box EEPROM data from the host
`CONFIG,S?;sync`	Get synchronization information
`CONFIG,SS,mode`	Set synchronization mode

Public data area

The public data area stores information packets. This area holds data such as the size of the expanded memory and settings for programmable signal conditioners. Packets to write to the EEPROM are built in the host memory and transferred by the `CONFIG` command using a mechanism similar to the `TO1401` command.

Public data packet structure

Byte	Contents
0	The packet size, including this byte. 0 marks the end of the list.
1	The packet type code. CED reserves codes 0-127. User codes are 128-255.
2...	The data for this packet starts here.

Public area code 0 — This 10 byte area recorded the start address and size of any expanded memory in the 1401*plus*. If this code is absent, expanded memory is not used. If you really need more information about this see archived versions of this manual.

Public area code 1 — This 4 byte area holds the number of ADC channels in byte 2 and bit 0 of byte 3 is set if there is a second sample and hold fitted.

Public area code 2 — This 4 byte area holds ADC trimming information for the micro1401. Byte 2 holds the gain value and byte 3 holds the offset.

Public area code 3 — No longer used. Was ADC information for the Power1401.

Read data for a known code

`CONFIG,RC,hoOff,max,code` is used to read back a public packet with a known code into host memory. If the block requested doesn't exist error 253,3 is returned.

`hoOff` The offset into the host transfer block. See `TOHOST` and `TO1401` on page 17.

`max` The maximum number of bytes to return (less may be returned).

`code` The code of the block to be read, returned in the data as the byte at offset 1.

Read the nth data packet

`CONFIG,RN,hoOff,max,n` reads the n^{th} public data packet to host memory. The `hoOff` and `max` fields are as above.

`n` The public data area to read in the range 1 to the number of public data areas. If a number outside this range is given, the command returns error 253,3.

Write public area packet

`CONFIG,WB,hoOff` writes a block of host memory to the public area. If a packet exists with the same code, it is deleted before the new packet is written. Error 253,3 is returned if there is not enough room in the EEPROM to hold the new packet.

`hoOff` This is the offset into the host transfer block as described above.

Read general top box information

`CONFIG,T,dev;type,rev,serial,tags,cpld` returns information about top box number dev (use 0 for motherboard information). The returned information is:

`type` The top box type code.

`rev` The top box revision. 0=revision A, 1=B and so on.

`serial` The top box serial number.

`tags` The count of stored EEPROM tags.

`cpld` The CPLD (firmware) revision.

Read one byte of top box EEPROM

`CONFIG,TV,dev,offs;val` reads a single byte from top box number `dev` at offset `offs` in the range 0 to 252 into the EEPROM. The result is in the range 0 to 255.

Transfer top box EEPROM data

`CONFIG,RT,hoOff,dev,offs,n` and `CONFIG,RT,hoOff,dev,offs,n` read and write the contents of a top box or motherboard EPROM. Data transfer is by a `TOHOST/TO1401` like block transfer mechanism using area 0. The command fields are:

`hoOff` The offset from the start of transfer area 0 in the host.

`dev` The top box number as 1 to 3 or 0 for the motheboard.

`offs` The start offset into the 256 byte EEPROM area.

`n` The number of bytes to read or write; `offs+n` must be <= 256.

Using the EEPROM private data header

The private header can be expanded without affecting the operation of the software. The standard header is 11 bytes long and has the following structure:

Bytes	Contents
0	Bits 0-2 hold the EEPROM size in bytes as 0=128, 1=256, 2=512, 3=1024, 4=2048, 5=4096, 6=8192 and 7=16384. Bits 3-7 are unused and set to 0.
1	The offset to the public area in the EEPROM (also the private header size). Do not assume the value 11; check the size to guarantee compatibility.
2	The functional level for software operations of 1401. The first release boards have this field set to 0.
3	The digital card revision level. Revision A=0, B=1 and so on.
4-5	These bytes hold the digital card serial number, least significant byte first.
6-9	These bytes are normally 0. When used, they hold 'permissions' to run software packages. Up to 4 permissions can be held without extending the private header. If more space is required this area is expanded. Code 255 flags a demonstration machine which may run any software package
10	This is a checksum for the private header. The sum of the private header bytes treated as unsigned numbers (including the checksum) is a multiple of 256. This is the last byte of the private header.

Reading the private header

`CONFIG,RH,hoOff,max` reads the private header from the EEPROM to the host memory. See page 17 for block transfer details.

`hoOff` The offset into the host transfer area 0 at which to transfer the private header.

`max` The maximum number of bytes to be transferred.

Writing the private header

`CONFIG,WH,hoOff` writes the private header. As the EEPROM operation depends on the integrity of this header, and the header holds information that can be used in software protection schemes, blocks offered as headers must meet certain specifications before the 1401 accepts them. These specifications are private to CED and are not published.

Error codes

The following errors are possible when using the `CONFIG` command:

Code	Meaning of the ERR command report
253,1	EEPROM reading or writing timed out, probably due to a hardware problem.
253,2	The public area of the EEPROM was inconsistent during power-up self-test. If this is seen after the `WB` command variant, the public area will only hold the new block. If the error occurs after the `RC` or `RN` commands then no data has been transferred and the error will recur until a `WB` command is used.
253,3	A requested public data area does not exist or there is not enough room to write a public data area.
253,4	The private header of the EEPROM is corrupt. The only sub-commands which will work in this state are the `RH` and `WH` variants.

Synchronization of 1401s

If your 1401 has a synchronization option (or the possibility of fitting one), there are two extra commands available. The synchronization option allows multiple 1401 to run at exactly the same speed, eliminating the possibility of time drift between multiple units due to the internal clocks running at slightly different rates.

Query synchronization mode

`CONFIG,S?;sync` returns the synchronization mode as -1 if the synchronization hardware is not fitted, 0 if it is fitted but the unit is not synchronized and as 1 if the unit is currently in synchronized mode.

Set synchronization mode

`CONFIG,SS,mode` sets the mode of operation. You can set `mode` to:

0 Automatic. This is the default mode. If an external synchronisation signal is detected, the unit will derive the clock from the external signal.

1 Force slave mode. The clock will only be derived from an external signal. If there is no external signal, there is no clock. Not recommended; used for testing purposes.

2 Force independent operation. Ignore any external synchronization signal.

Appendix A: 1401 family differences

Standard 1401 and 1401*plus*

This section gives you the information you need to convert a standard 1401 program for the 1401*plus* making the minimum changes to your application. Re-build your program with the latest version of the 1401 family language support libraries. In most cases, you will find that your 1401 programs will now work equally well for both 1401 and *1401plus* (and micro1401). You may still notice the following differences:

1. If your program used a mechanism other than `Ld` to load commands, or set the file extension for commands to '.CMD' explicitly, use `Ld` and remove any file extensions.
2. If you use `FFT`, `GAINPH`, `ADDPWR` or `DLOGPWR` all these commands are held in `FFT`. So for any or all of these commands, load the `FFT` command. To be compatible with the standard 1401 use a `Ld` command with `FFT` first followed by the other commands you need (`Ld` tests if a command is loaded before searching on disk).
3. `MEMTOP` always returns 65534,0 in the 1401*plus* unless the user data space is less than 65534 bytes. In the standard 1401, `MEMTOP` typically returns 50000,1536 with the difference between the two figures being the user data space. Thus well-behaved programs that ask the 1401 how much user space is available have more space in 1401*plus*. This may cause problems with programs that allocate memory in the host based on the user space in the 1401 as the 1401*plus* will have more free space.
4. If you load a large number of commands (more than 24 kB of commands) you may run out of command space. See page 18 for expanding command space.
5. When you load a new command, the user space does not decrease in the 1401*plus* (as is the case with standard 1401). This should not be a problem!
6. The 1401*plus* has more error codes (you do check error codes don't you?). Programs which check for `ERR` returning 0 or non-zero will work without any problem.
7. The 1401*plus* normally has much more memory space (typically 900 kB or more) for user data, even though `MEMTOP` reports 65534 bytes. All commands that use the user memory support the use of much larger numbers for the start and size of a region. This means that commands that would cause standard 1401 errors may not do so in the 1401*plus*. The following would always cause a standard 1401 error (no standard 1401 has 60000 bytes of user space), but would run correctly in a 1401*plus*.

   ```
   ADCMEM,I,2,40000,20000,0,1,C,10,10
   ```
8. The 1401*plus* is much faster. This is especially true of arithmetic calculations where the 1401*plus* can be 40 times faster than the 1401. If a program ran round a loop, performing a background task until the 1401 had finished a task you may find that with a 1401*plus* the background task never has a chance to be done!
9. The MassRAM can be emulated on a 1401*plus* with expanded memory (4 or 16 MB). If your program asks for the size of MassRAM, it may be surprised if it did not allow for 16 Mb. See the `MEMTOP` command for information on MassRAM emulation.
10. The standard 1401 returns numbers in the range -32768 to 65535. The 1401*plus* can return numbers from 2^{31}-1 to -2^{31}.
11. Very old programs that used negative numbers to represent the number range 32768 to 65535 will fail with 1401*plus*. Use positive numbers.

We provide all the standard 1401 commands described in the (old) CED 1401 Intelligent Interface Programmers' handbook (except `YT` which is no longer supported and `AUDMR`). We have also translated some obsolete 1401 commands (`ADCMEMI` and `ADCMEMF`) to allow old applications to be easily upgraded. We urge users not to write the obsolete commands into new programs but to use the modern equivalents (`ADCMEM` and `MEMDAC`) and ideally to convert their old programs to use the modern commands.

micro1401 differences from 1401plus

This section describes the differences you will see between a 1401*plus* and a micro1401 as a programmer. The first thing to say is that there are not very many! Although the two units look very different and use different processors, they behave in a very similar manner (and share quite a lot of code internally).

1. micro1401 commands have the extension `.ARM` on PC systems and have a type of 1403 on Macintosh systems. As long as you link with a recent 1401 family interface library you will see no difference.
2. The `ADCBST` command is included in `ADCMEM`, it is not a separate command on disk. If your program only uses `ADCBST` you can always rename `ADCMEM.ARM` to `ADCBST.ARM` on DOS systems. On a Macintosh you must change your code.
3. `ADCPERI` is no longer supported. `PERI32` has been the preferred command for several years and has all the features (and more) of `ADCPERI`.
4. There are no MassRAM commands. The micro1401 does not emulate or support the MassRAM. Just about anything you could do with the MassRAM you can do in normal memory. `ADCDAC` can be run as `ADCMEM` and `MEMDAC` simultaneously.
5. Commands that use 16-bit and 32-bit data require that the data arrays are aligned to a 2 byte and 4 byte boundary. You will get an argument error (at the `st` argument) on any non-aligned memory use.
6. There are 2 DACs and 4 ADC channels on an unexpanded micro1401. Rather than give an error, to be more compatible with 1401*plus* the micro1401 connects ADC channels above 3 to an internal ground unless an expansion unit is present.
7. There is only one clock F input, on the rear connector, not one for each clock.
8. If you have a second sample and hold, it is on channel 3, not 7.

Power1401 differences from micro1401

If your code works with a micro1401, you have very little to do to make it work with a Power1401.

1. Power1401 commands have the extension `.ARN` on PC systems and have a type of 1404 on Macintosh systems. As long as you link with a recent 1401 family interface library you will see no difference.
2. There are 4 DACs and 16 ADC channels, and the analogue system is 16-bit (rather than 12-bit in the micro1401). There is no provision for a second sample and hold on the main board. There is a gain option for the ADC.
3. It is very much faster, which may cause you problems.

Micro1401 mk II differences from micro1401

If your code works with a micro1401, you have very little to do to make it work with the Micro140 mk II.

1. Micro1401 mk II commands have the extension `.ARO` on PC systems and have a type of 1405 on Macintosh systems. As long as you link with a recent 1401 family interface library you will see no difference.
2. Clock 0 and 1 prescalers, and all stages of clocks 2, 3 and 4 can be set to divide by any number in the range 1-65536. This does not require code changes unless you want to take advantage of this.
3. The external clock input for clock 4 is ADC Ext, not the rear panel F input as is the case for the micro1401 and Power1401.
4. There is provision for DAC expansion. Up to 6 extra DACs can be fitted.
5. There is provision for more memory (most units have 1 MB but 2 MB is possible).

Micro1401-3 differences from Micro1401 mk II

If your code works with a Micro1401 mk II, you have nothing to do to make it work with the Micro1401-3 unless you check the 1401 type.

1. Micro1401-3 commands have the extension `.ARQ` on PC systems and have a type of 1406 on Macintosh systems. As long as you link with a recent 1401 family interface library you will see no difference.
2. The processor is some 20% faster, but this should not be a problem.
3. The DAC outputs are 16-bit, but this should not affect your code.
4. You must use a USB interface.
5. There is provision for more memory (all units have 4 MB but 12 MB is possible).

Power1401 mk II, -3 differences from Power1401

If your application code works with a Power1401, it will work with later versions of the Power. Changes are:

1. Power2 commands have the extension `.arp`, Power3 commands have `.arr`. As long as you link with a recent 1401 family interface library you will see no difference.
2. There is only the USB interface, but this is much faster than the CED parallel interface.
3. These units have more memory and are faster, but this will not normally cause any problem in existing code.

Using new features

If your program wants to take advantage of the features in a particular member of the 1401 family, you must first determine that you have one! The `Get1401Info` function provided with the 1401 language support can be used to get information on the type of 1401 as well as the revision level of the 1401 device driver, the type of host computer and the current state of the 1401. The main new features that impact applications are the availability of a much larger user data space, and much greater speed of operation.

Index