THRUSTMASTER®

HOTAS COUGAR

REFERENCE BOOKS

Version 1.4b

INTRODUCTION

Greetings!

Well, first of all, CONGRATULATIONS and THANK YOU for investing in the Thrustmaster HOTAS Cougar! The massively powerful, ruthlessly precise controller you now have in your hands is the proud result of a full two years of dedicated studies and research, led by the profound wish to create a cutting-edge simulation game controller that would match the most demanding gamers' expectations. And here we have it – behold the proud heir of the FLCS and F-22 PRO series! However, much as the challenge of creating a successor worthy of the previous series of controllers sounded like fun to all concerned, successfully bringing this deity to life proved to be a lot more than just… erm, challenging – and that's an understatement!

When this insane project first began, Thrustmaster was faced with two choices: to effectively just upgrade the existing sticks, retaining and expanding their existing functionalities, or, to start again from scratch, and invent a completely updated, advanced product. Obviously, by choosing the latter, Thrustmaster had also chosen to follow the hardest possible path. For what unfathomable reason? Quite simply because whenever Thrustmaster had chosen to release a new HOTAS controller, the new joystick had always taken quality to a higher level, and set every limit far beyond what the standard at the time had to offer – so, with the HOTAS Cougar, the time was nigh to redefine what a true-to-form hardcore joystick was.

The result of endless brainstorming sessions and sleepless nights is this highly realistic, incredibly versatile, awesomely flexible and monumentally potent controller – beyond all doubt, with the jaw-dropping collection of yet unmatched features such as interchangeable handles, advanced (yet deceptively easy) programming capabilities, dead simple plug'n'play connectivity, added to the outrageous number (and weight) of die-cast parts make the proud HOTAS Cougar the real deal for the years to come.
This brute is also a success for the whole simulation-addicted gaming community – believe us, this deity is no cheap, fancy, lightweight contraption optimised for the coin-ops of yore, but a hardcore extravaganza F-16 control replica, designed for those who demand nothing short of the best - truly the ultimate high-end flight sim controller that will leave both hostiles and rivaling flyers wishing they had opted for a career as ground personnel!

Of course, this beast would never have seen the light of day without all the support and encouragement we received from people we met at shows, who chatted with us on forums and had lengthy e-mail discussions with us – so, to all you guys out there, thanks for being so supportive to us! And let us not forget the people out there who really deserve some extra special thanks from us; first of all, the staff at Thrustmaster/Guillemot who worked on this project, and the beta testers, who did a great job hunting all those bugs down, and experimented with this hazardous equipment at the risk of their lives… Also, thanks to our friends and families, who put up with us all the time we were working on this project!

Ok, enough romance for now, let's get real!

If you are experienced at all with the previous range of Thrustmaster flight simulation gear, the power of the HOTAS Cougar will leave you feeling awed – and yet, also quite at home. In fact, some features may actually *look* the same, but don't be fooled – it's all new material in here, and as an acquainted flight sim user, you, better than anyone else, will be the person to fully assess all the hard work we put into this to provide you with unheard of new might.

With this new release, we have improved and innovated on all points - mechanical, electronic and software components. Whereas providing a more user-friendly controller than the seven year-old joysticks was not to prove to be excessively difficult, what we have done is chosen to give every gamer out there, and even newbies, the ability to make the most out of this highly resourceful brute. Indeed, the release of Microsoft Windows has made the introduction of HID-compliant controllers a reality – plug'n'play joysticks that don't require any particular configuration to operate. Well, this is exactly what the HOTAS Cougar has in store for you – just plug it in, and play your favourite games… It's that simple.

Well, that's cool news - but a lot of *you*, who have bought the HOTAS Cougar for the advanced programming capability it claims, are expecting a lot more from *us* than just a *great* controller. So check – here it comes! To match the unequalled precision it provides, the HOTAS Cougar offers all the same programming features as the F-22… and much, much more, as you will learn by reading through this Reference Book. The programming features we will be broaching hereinafter are in fact so powerful and comprehensive, that they will enable you to optimise any of your programs to make the most of each game, or even to correct bugs or missing functions in a simulator. So, the best way to go about discovering them and taking them all in (that is, without incurring a massive migraine and code-induced nightmares) is simply to read carefully and attentively through this literary milestone of a book, and give yourself the time to think, "OK, now what am I expected to do with *this*?"

As outlined above, this document is what we refer to more as a “Reference Book”, rather than your usual run-of-the-mill “Manual”. Anyone who's been used to or seen the manuals for the original F-16 FLCS, TQS and F-22 PRO controllers will appreciate the difference between them and this reference book. It contains everything you need to know from setting up your controllers, using the Cougar Control Panel software, learning the basics of Cougar programming right through to detailed information on every aspect of the programming statements that set the HOTAS Cougar aside from other programmable controllers. Backing up this manual are highly detailed help files, wizards, tutorials and other useful and intuitive applications within the programming software. At the end of the day, we think we've provided the most comprehensive documentation that has ever shipped with any controller.

Be under no illusions about the HOTAS Cougar and this reference book - this is a hardcore, state-of-the-art controller. And this reference book does have sections and statements that you may want to read over more than once. But one of the reasons why this manual is so exhaustive is that we have been acutely aware that with the previous TM controllers, people found the manual too brief and hence the controllers too difficult to get into. You will therefore find that this manual is very easy to read, and introduces everything gently, and simply. It's very kind on those of you who just want some basic knowledge about a statement or feature, as well as providing comprehensive and detailed information for those that want it. Whatever your level, we've got you covered :) So, if you’re using the HOTAS Cougar for the first time, we definitely suggest that you take the time to read through the first sections in this book, or you will very likely never be able to make the most of the HOTAS Cougar. The HOTAS is a very flexible programmable device indeed, leaving you several ways and offering many tools to attain one same result - but beware, it nonetheless boils down to programming, and on these grounds, the more logical and methodical you are, the better you will fare.

Without further ado, well - it’s over to you now! Read on, make the most of this ultimate controller, and show those hostiles what the HOTAS Cougar is about!

For a Spanish translation of this book, please visit:
http://www.escuadron111.com

For a French translation of this book, please visit:
http://www.checksix-fr.com/

For a Dutch translation of this book, please visit:
http://thrustmaster.vanree.net/

For a German translation of this book, please visit:
http://www.thrustmaster-x-files.de/

For a Russian translation of this book, please visit:
http://www.hotas.ru

ACKNOWLEDGEMENTS

Our deepest thanks goes out to the following people and websites for all their help and support in the course of this massive project – cheers to you all! ☺

Beta testers

Olivier "Red Dog" Beaumont
Robin "Emacs" Breyl
Jan-Albert "Anvil" van Ree
James "Nutty" Hallows

Company and squadron

Mark “Frugal” Bush & frugalsworld.com
Wingmen-alliance.com
Escuadron 111.com
Microsimulateur
SimHQ.com
Sim-arena.com
Aimsworth Coproration
Combatsim.com
Ubi Soft
Checksix-fr.com
Dogfighter.com
Desktopsims.com
Gamekult.com
Fast jet Flight Simulation (a.k.a. HAM technologies)

Special Thanks

Len "Viking1" Hjalmarson
Matt Wagner
James R Campisi
Flavien "Vox" Duhamel
François Pimenta
David "Micro" Vely
Philippe "Twech" Dezeure
Philippe "Jag" Dubois
Lew/+Silat
Rob Coppock
Laurent Espinasse
Fernando Oscar Garcia Minguillán
Guillaume "Ghostrider" Houdayer
Oleg Maddox
Jim Staud
Jean-Dominique "Bing" Belin
Emmanuel "Judy" Durant
Thomas "Doloop" Coulomb
Denis "Dugin" Blary
David "Zip" Pierron
Ulf Muckel
Hal Bryman
Jose "Oso" Benito
Stanislav "huMMer" Vartanian

THRUSTMASTER

HOTAS COUGAR

CONTROL PANEL (CCP) REFERENCE BOOK

Introduction

The HOTAS Cougar is a virtually driverless joystick solution, that does not require any programs to run in the background to apply different axis settings, or to perform emulation of any sort. For this reason, it is important that the joystick itself is aware of all changes being made to the axes, and therefore, other applications such as CTFJ (Bob Church's Centering Tool For Joysticks), and the Windows Calibration routine should not be used - if they are, many advanced changes to the axes' calibration will produce varied results.

PLEASE NOTE: DO NOT USE WINDOWS CALIBRATION TO CALIBRATE THE HOTAS COUGAR. INSTEAD USE THE HOTAS COUGAR CONTROL PANEL TO MANUALLY CALIBRATE THE HOTAS COUGAR.

The HOTAS Cougar Control Panel (CCP) application is used to change various parameters for the output of the HOTAS Cougar, ranging from the mapping of axes to the different output modes of the Joystick (emulation mode, Windows mode…). What follows in this reference book is a button-by-button explanation of the application, and how you can expect changes applied to these buttons to be reflected in the operation of the Joystick.

To display the settings that are currently active in the Joystick, open the HOTAS CCP while the Joystick is connected.

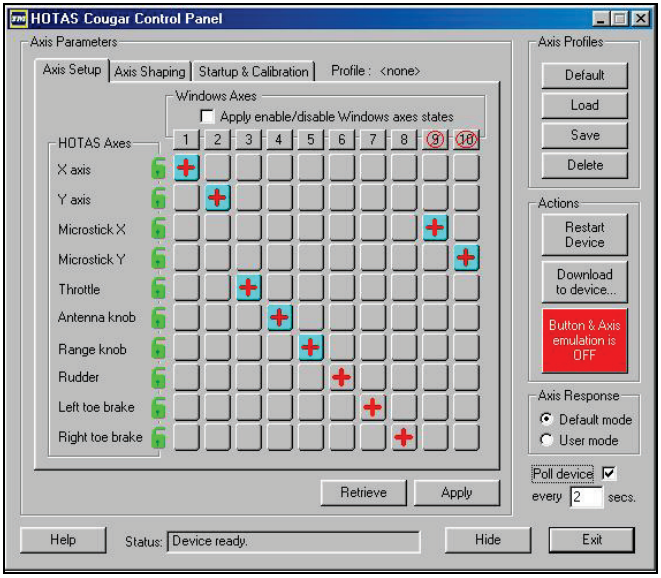


Figure 1: HOTAS Cougar Control Panel in the default configuration

Axis Profiles

Profiles are used to quick-load previously defined configurations into the HOTAS CCP and Joystick. In the Profiles section of the HOTAS CCP, four buttons are available: “Default”, “Load”, “Save” and “Delete”, all of which are explained in the sections below.

DEFAULT

Clicking the Default button displays the Default parameters which are used by the Joystick in Windows mode. This includes all axes mapped according to DirectX standards, most axes in the positive defined direction, upper and lower dead zones of 5%, centre dead zones of 7%, linear curve settings (0), base of curve centred, and the “apply enable/disable Windows axes states” disabled. If the Joystick is connected and the screen display does not make any sense, the best course of action is to click on the Default button, and perform the changes from there.

SAVE

Clicking on the Save button saves the current configuration, which has been defined in the Axes Parameters section, to the HOTAS/Profiles folder, with the file extension ’.TMC’ (Thrustmaster Configuration). If any Calibration has been performed, it will also be saved in this file. Saving profiles this way means that different axes parameter preferences can be saved for different games, and can be loaded for future use.

LOAD

Clicking the Load button loads the saved configuration profile into the graphical display (the tabs). This does not however load the file into the Joystick; performing this action still requires you to click on the Apply button, described further on in this reference book.

DELETE

This button is used to delete a profile.

Joystick Modes

The HOTAS CCP allows you to set the Joystick modes pertaining to the Axis Parameters, Calibration Options, and Emulation features. Setting these Joystick modes does not require you to click on the Apply button, as any change in the Axis response mode will automatically instruct the Joystick to change modes.

AXIS RESPONSE

In Default mode, the Joystick will use the default axis setup and shaping data. This data is always loaded into the Joystick, and is used whenever the Joystick is connected - with the exception of the numbered axis buttons, should the last download have specified to enable the visible status (this is further explained in the "Changing Windows Axes States" section). When the Axis Response feature is set to User Defined Mode, the Joystick will use the following data, as you specified:

- Axis Mapping
- Reversing Data
- Locking Data
- Curve Information
- Base of Curve
- Dead Zone Information
- Trim Settings

For information on the Calibration options, please see the section on Calibration. For information on the Emulation modes, please see the section on Button & Axis emulation.

Axis Parameters Section

The HOTAS CCP application features three tabs: the Axis Setup Tab, the Axis Shaping tab and the Startup & Calibration tab. Performing any changes to either of these three tabs will not cause the associated action to be performed immediately by the Joystick; indeed, any changed parameters must first be applied to the Joystick, and the Axis Response must be set to User mode before the Joystick will use this new information. To better follow with the explanations given, first open the HOTAS CCP, and click on the Default profile button.

AXIS PARAMETER BUTTONS

The Axis Parameter section (excluding the aforementioned tabs) contains two buttons: “Retrieve” and “Apply”. Their functionality and uses are explained in the following sections.

Apply Button

The Apply button allows you to download the configuration parameters described in the Axis Parameters section to the Joystick. Until you actually perform this operation, the Joystick has no knowledge of any changes applied in the Axis Parameters tabs, and the Apply button will download whatever parameters are currently displayed or defined in the tabs. For the Joystick to be able use this downloaded information, the Axes Response must be set to User Mode, which is performed automatically by clicking on the Apply button.

Retrieve Button

The Retrieve Button is used to upload the configuration currently saved in the Joystick. If any of the parameters are changed in Emulation mode, these can be reflected by performing a Retrieve operation, and by checking the appropriate section. The HOTAS CCP application automatically performs a Retrieve operation on startup, to display the Joystick’s current configuration; if the Joystick is not connected, an error message is displayed, and the Default profile is shown. By clicking on the Retrieve button, you can also view the Joystick’s current mode.

AXIS SETUP TAB

Several functions are available on the Axis Setup Tab, and their respective purposes can be summarised as follows:

- Changing Axis Setup
- Reversing the direction of an axis
- Locking the values of a particular axis
- Changing the axes that are recognised by Windows

These functions are explained in detail in the respective following sections.

Changing the Axis Setup

If you need a particular axis directly to control a different axis in a specific configuration, the easiest way to accomplish this will be to change the axis' setup. When loaded in its default format with only the TQS (Throttle) connected, the axis setup will appear as such:

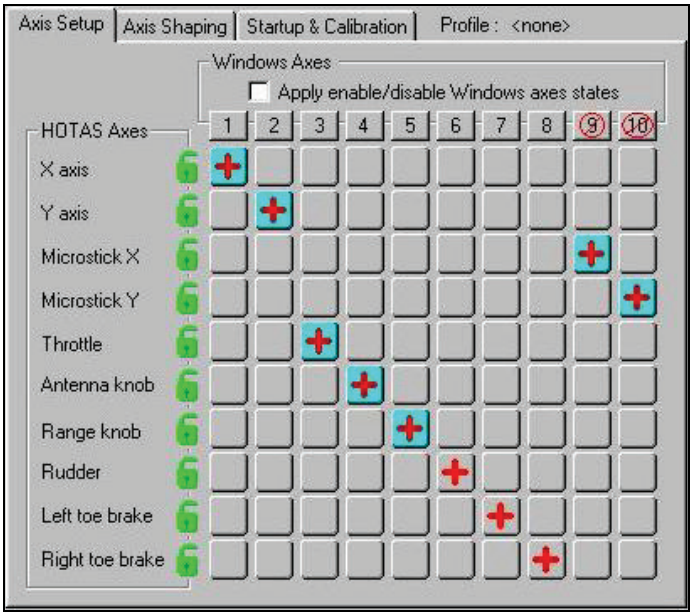


Figure 2: Axis Setup Tab in Default configuration

The numerous buttons represented in the centre of the Axis Mapping tab show the currently selected mapping configuration; this shows that the X axis is mapped as the first axis, the Y axis as the second, the Throttle axis as the third, etc… If we want the second axis (the DirectX Y axis) to be controlled by the throttle (which is often quite handy for racing games), we only need to click on the Throttle's row in the second column, and the window will change appearance to look like this:

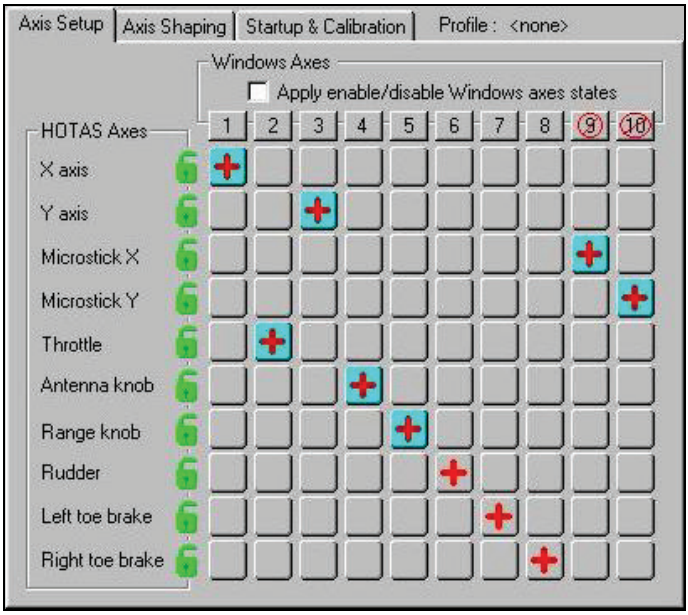


Figure 3: Axis Setup Tab with the Throttle knob as the Y Axis

As can be seen, the Throttle axis has been assigned to the second position, and the Y axis has taken the Throttle axis's place as the third axis. All axis swapping operations can be performed this way, and although windows will "see" the axes in their changed state, all axis programming inside the joystick - including all reverse, curve, dead zone, and emulation programming - will remain specific to the physical axis; therefore, if the Throttle is programmed to have a precise curve, and is then swapped with the Y axis, the Throttle will control the Y axis in exactly the same way as it controlled the Throttle axis.

Notice that the background of the checked buttons in the sixth, seventh and eighth column are grayed, as opposed to the others (which are light blue).
This is because in this configuration the RCS (rudder and toe brakes) are not connected. The buttons with the light blue background indicate the axes which are physically connected. To have the X and Y axes controlled by the Microstick, the Axis Setup tab would look like this:

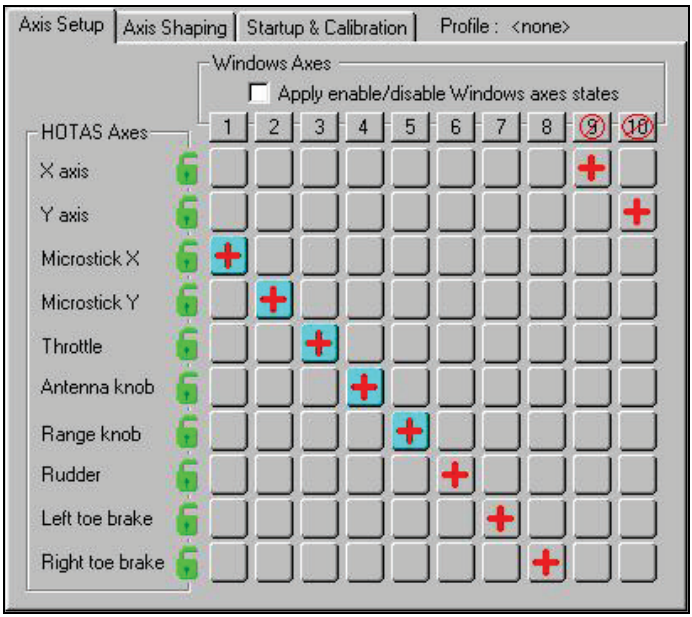


Figure 4: Axis Setup Tab with the Microstick controlling the X & Y Axes

Reversing The Direction of an Axis

To reverse the direction of an axis, click on the desired button with a plus sign (or minus sign) and it will toggle the direction of the axis. In the case where you would like to change the direction of the Y axis, the appearance of the Axis Setup tab would be as follows.

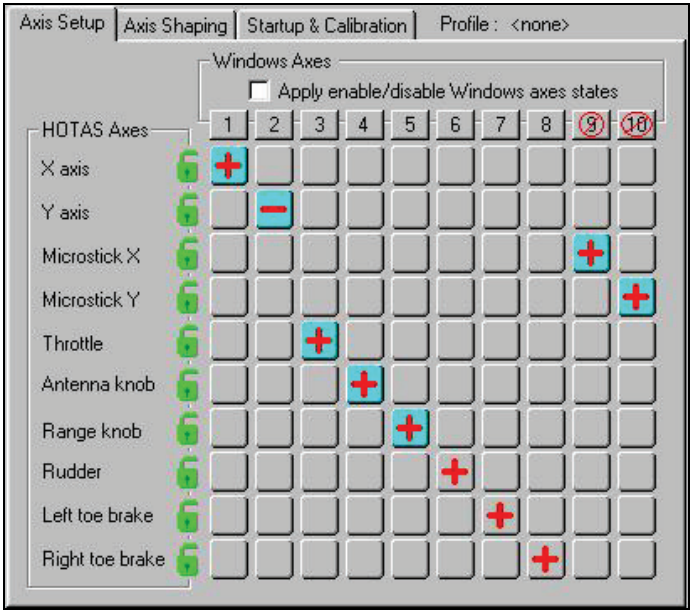


Figure 5: Axis Setup Tab with Y Axis reversed

Locking an Axis

On the right side of each axis name is a lock-shaped icon. If this "lock" is open and is green in colour, then the axis is "unlocked" and will behave normally. If the lock is closed and red in colour, then the corresponding axis is "locked", and its value cannot be changed. To toggle the state of the lock, click on it, and it will switch between the two states described above.

Changing Windows Axes States

This title may not accurately reflect this section's utility, but without getting too technical, it's still the best description of the following section's function. Notice that a button is located at the top of each column, and in Default mode, buttons 9 and 10 have a red not-circle in front of the numerical text – these buttons are depicted below:

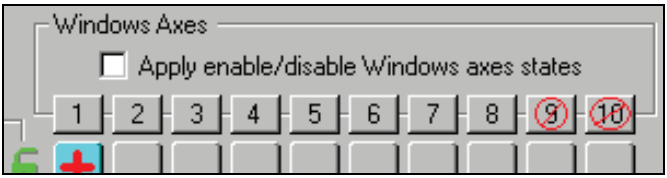


Figure 6: The "numbered axis buttons" located on Axis Setup Tab

Each of these buttons represents a different DirectX axis, these axes being:

Axis Number	DirectX Axis Name
1	X Axis
2	Y Axis
3	Z Axis
4	Rotational X Axis
5	Slider 0
6	Rotational Z Axis
7	Slider 1
8	Rotational Y Axis
9	<none available>
10	<none available>

Table 1: Axis Numbers to DirectX Names

In addition to the fact that you cannot enable the ninth and tenth axes, it is also impossible to disable the first and second axes, since the Joystick drivers' programming implies that a Joystick features at least these two axes. If you click on any of the buttons numbered 3 through 8, the red not-circle will be toggled on and off. When the Joystick is connected, the names of the axes, as well as the number of axes as "seen" by Windows, are determined by one of two possible sources:

1. The actual physical setup of the Joystick
2. The states of these "numbered buttons"

Explanations of these two points follow on the next page.

AXES UNDER PHYSICAL SETUP

What do we mean by the "physical setup" of the joystick? All this means is what other controllers (e.g. throttle, rudders) are connected to the joystick, which in turn defines how many axes Windows detects the Cougar as having. 6 physical setups are available for the HOTAS.

1. Joystick is plugged in alone
2. Joystick is plugged in with the TQS attached
3. Joystick is plugged in with the regular 1 axis RCS attached
4. Joystick is plugged in with the new 3 axis RCS attached
5. Joystick is plugged in with the TQS and regular RCS attached
6. Joystick is plugged in with the TQS and new RCS attached

Each of these possibilities will display a different combination of axes for the HOTAS inside of Windows. Below is a table representing all 6 of the configuration possibilities listed above, and displaying which axes will be present for each.

CONFIGURATION	AXIS NAMES							
	X	Y	Z	Rx	SL0	Rz	SL1	Ry
1	●	●						
2	●	●	●	●	●			
3	●	●				●		
4	●	●				●	●	●
5	●	●	●	●	●	●		
6	●	●	●	●	●	●	●	●

AXES UNDER "ENABLE WINDOWS AXIS STATES" SETUP

Using the numbered buttons, you can change the axes which will be recognized by Windows. The procedure to enable use of these buttons is listed below.

1. Select the desired axes which you would like Windows to "see" by toggling the state of the individual numbered buttons (i.e. enabling or disabling them), until the required configuration is reached.
2. Make sure that the "Apply enable/disable Windows axes states" check box is selected.
3. Load the file into the Joystick by using the "Apply" button, and click 'OK' to perform a restart of the Joystick.
4. Now, the Joystick will be "seen" by Windows, featuring the axes you specified.

The point of this is that even if you do not own Rudders, or the new Rudders set (with toe brakes), we can make DirectX "believe" that either of these devices are present, and then control their input values with any one of the available axes, or even through an emulation file.

It is important to note here that the Joystick will load in the user-defined "Enable Windows Axes States" mode as long as the last file downloaded to the Joystick had the "Apply enable/disable Windows axes states" check box activated**.**

AXIS SHAPING TAB

The Axis Shaping Tab contains controls to modify more advanced Joystick configuration features, and is very useful when it comes to enabling users to tune each axis according to their own personal preferences. The "Axis to set" combo box at the top of the tab selects the desired axis, among the ten possible axes, whose properties you want to adjust. Once a parameter has been changed, the associated change is graphically displayed on the graph located to the right of the list of different parameters. The change in display is triggered by either switching parameter categories, or by clicking on the Refresh button located to the top right of the tab. The various adjustable parameters are as follows:

- Upper Dead Zone (UDZ)
- Lower Dead Zone (LDZ)
- Center Dead Zone (CDZ)
- Calibration Center
- Axis Trim
- Curve Settings (factor and base)

The parameters listed here can be enabled by selecting different Joystick output modes.

Dead Zone Information

By changing the Dead Zone Information (UDZ is Upper Dead Zone, LDZ is Lower Dead Zone, CDZ is Center Dead Zone), you can alter the inactive regions on each axis. The effects of successive changes will be reflected in the graph to the right-hand side of the area where the parameters are listed, dead-zone areas being highlighted in red. All values have a maximum of 100%, where 100% is approximately 30% of the actual axis movement on the Joystick. For instance, the default value used internally by the Joystick is 1% of each of the dead-zone areas, which gives a total of 3% of the complete axis travel. Below is a pictorial representation of this tab:

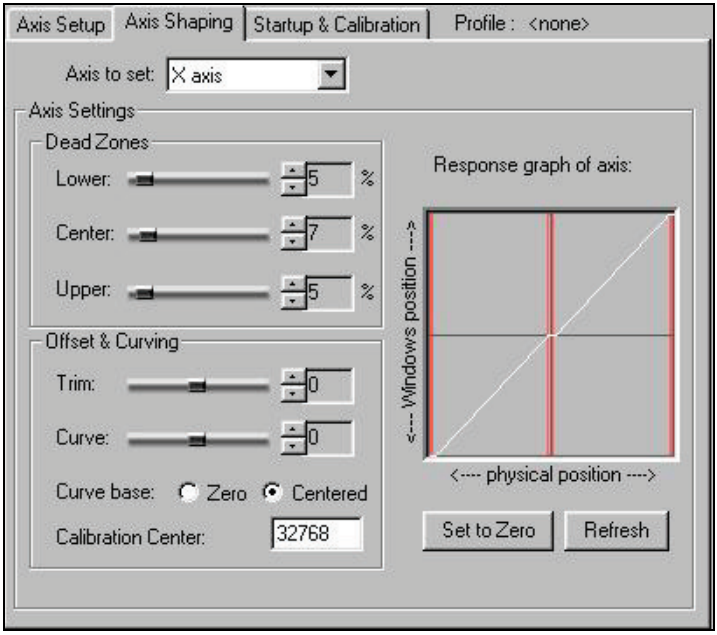


Figure 7: Axis Shaping Tab with Default Parameters

In the above figure, small red regions are apparent to either side as well as at the center. This plot shows the relationship between the physical position of the joystick axes, and the actual values that will be provided to Windows, for use in games and on your desktop. For reference, the horizontal axis of this graph represents the values that the Joystick collects from its axes, with the minimum value displayed on the left-hand side, in this plot. The vertical axis of the graph is

the actual output from the Joystick as it will be "seen" by Windows, with the minimum value displayed at the bottom, in this plot. Notice that in the dead-zone regions, the graph becomes a straight line, meaning that the output will be the same for various inputs. The lower dead zone is located on the left-hand side of the graph, and is a horizontal line; the upper dead zone is positioned on the right-hand side of the graph, and is also a horizontal line. The next figure illustrates what happens if we increase the CDZ value:

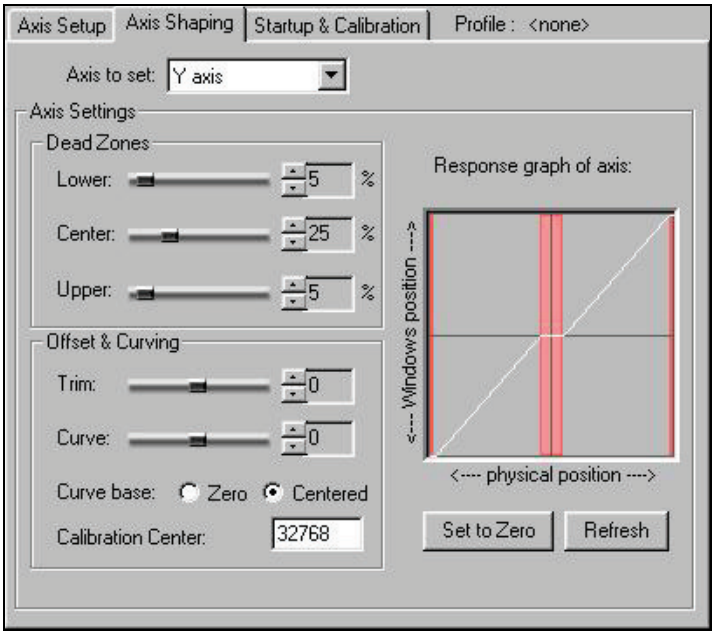


Figure 8: Axis Shaping Tab with Y axis CDZ increased

Figure 8 clearly shows that raising the Center Dead Zone value from 7% to 25% has proportionately increased the red highlighted region at the center of the horizontal axis, and the white horizontal line at the center represents the region of the axis where the Joystick will be inactive. The upper and lower dead zones react similarly.

Calibration Center

This value is the position that the Joystick axis will consider to be its central position. If the specified Center Position value is lower than the Joystick's current physical central position, then the axis will appear to reach a higher value when the Joystick is at rest in its central position. A similar result could also be attained by adjusting the Trim value in order to offset the physical values that the Joystick outputs; the Trim function's main advantage over the Center position feature is that Trim can be altered during emulation.

Axis Trim

The Trim function is used to offset the actual physical value of the axis to a new value. For example, if the Joystick axis is at a physical position equaling 30%, whereas Trim is set to –20%, then the actual position understood by the computer will be 10%. The maximum Trim setting is +/- 50%; this means that from the center position, it is possible to have the axis go from one end to another by setting the Trim. This is demonstrated in the figures to follow.

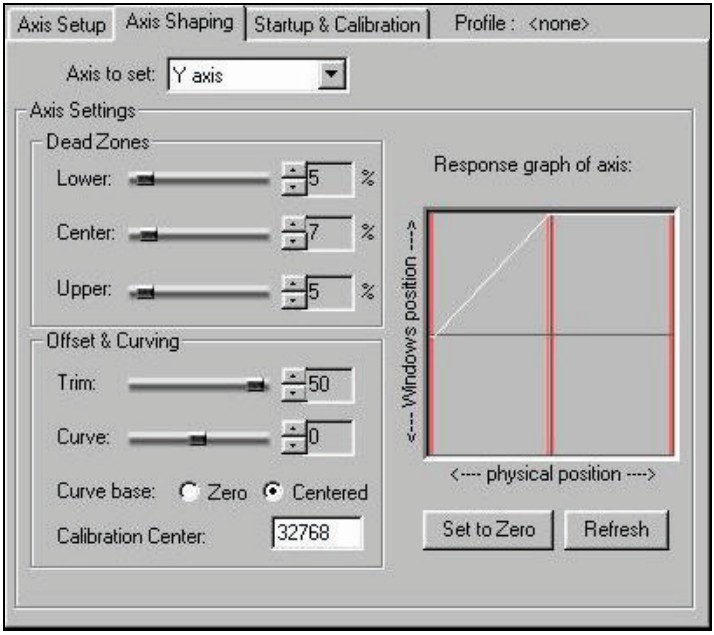


Figure 9: Trim setting at maximum

With Trim applied to the Y axis, the reaction can be observed immediately. With the axis at its physical center (the horizontal center), the output data that Windows receives is the maximum Y axis value. If the Y axis is moved in a direction increasing its value, there will be no reaction; if the Y axis is moved away from its physical central position down to its minimum position, the computer will observe that the axis' values are decreasing, until they reach a minimal output value, equal to the axis' normal center position.

If we were to reduce Trim to its minimum value, the graph would look as follows:

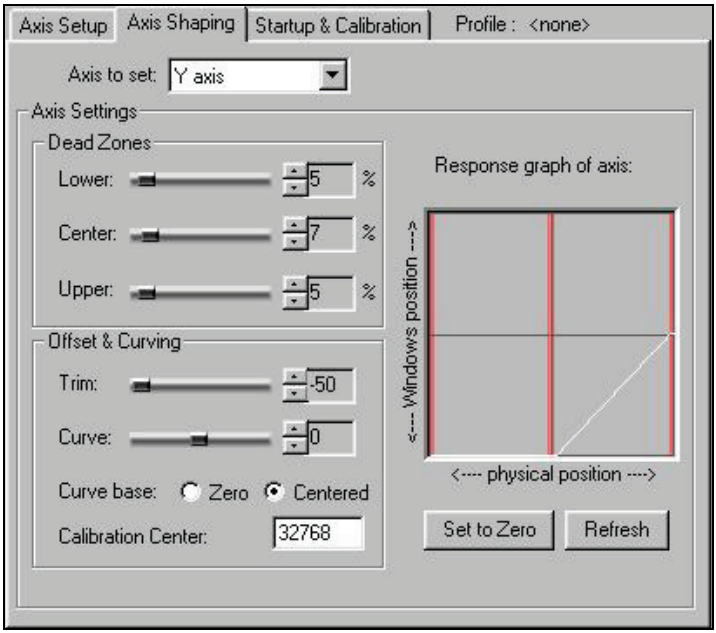


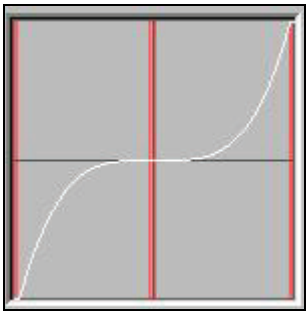
Figure 10: Trim Setting at minimum

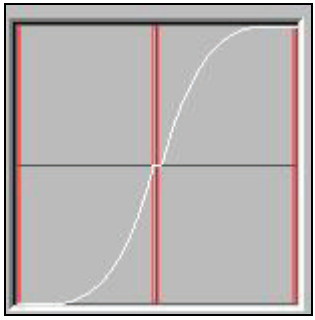
With this change applied, the Y axis value will be read as being minimal when the Y axis is physically at its central position, and will be read as being at its center when the Y axis physically is maintained at its maximum position.

Curve Setting

This setting can be adjusted to have the axis respond with a more exponential feeling, i.e. a “curved” feel, instead of a regular linear feel. The Curve parameter can have values between -32 and 32; however, setting several axes to such high values will significantly slow down the operation of the Joystick. Notice that values greater than 20, or smaller than -20 are not useful, and that the change in the curve is not sufficient to even justify using such values.

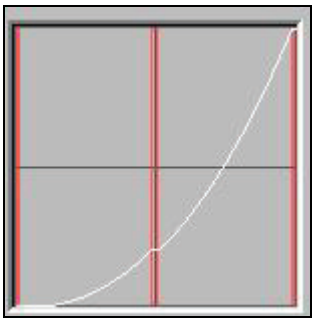
We will first take a look at curves with a centered base - so what might be the difference between a positive curve and a negative curve? Let’s look at the response when curve settings of positive and negative 10 are applied.

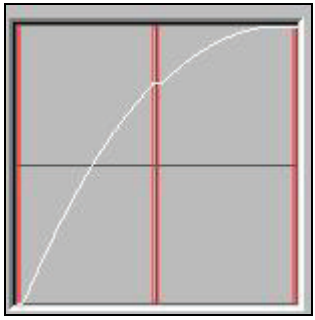




The figure to the left represents the axis with a Curve setting of -10 (negative); as can be seen, around the central position, very little change in the axis - almost like increasing the central dead zone. The rate of change then increases quickly, until the axis’ slope becomes almost vertical. This means that the Joystick will offer enhanced responsiveness around the limits of this axis’ travel. The figure to the right shows a graph of the axis with the Curve setting at 10, which appears to almost be the previous setting reversed; the axis is much more responsive around the physical central position, whereas at the extremities of the axis travel, responsiveness is far lower - almost like increasing the upper and lower dead zones.

The idea behind the 'Base of Curve' radio button is that although the response curves of the Joystick and Microstick axes should be curved, as we saw in the previous example, a full range ramp may however be more appropriate for axes such as the toe brakes and throttle. Below are the graphs for Curve settings of positive and negative five (as a value, ten is excessive), but with the Base of curve setting set to Zero.





The figure to the left represents the graph for an axis with a negative 5 Curve setting - notice how similar it is to the upper right section of the graph representing the axis with a Curve setting of negative 10 and a centered Base of curve. The Joystick will now react very slowly at the lower extreme of the axis, as if an increased lower dead zone had been added to the axis, and its responsiveness then increases until it reaches a peak at the upper limit of the axis travel. To the right is a Curve with a setting of positive 5, its base being Zero-based. Once again, notice how similar the entire graph for the Zero base positive curve is to the upper right section of the Center-based positive curve. Responsiveness increases near the physical minimum of the axis, and decreases around the maximum. The upper limit of the axis looks as though we have an increased upper dead zone.

STARTUP & CALIBRATION TAB

The startup and calibration tab contains information about how the Joystick will behave when the computer starts up and calibration configuration. This tab is separated into two sections, the top part is the startup options and the bottom part is the calibration configuration. Each of these sections is described below.

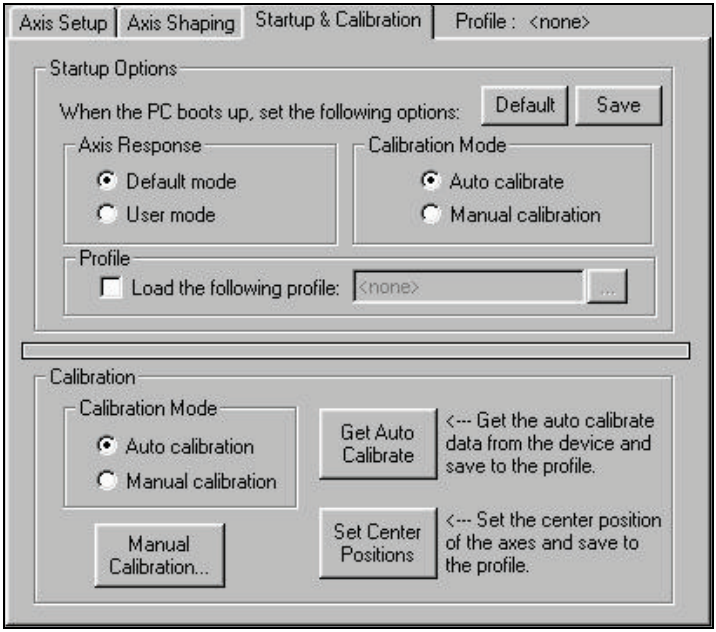


Figure 11: Startup & Calibration Tab

Startup Options

There are three options that the Joystick can be set to automatically when the computer starts up. These options are the Axis response, the Calibration mode and to load a specific profile. The Joystick, by default, will load in default axis response mode, auto calibrate mode and with the last profile that was downloaded to the Joystick. To change the default, click on the desired options and click on the 'Save' button. To choose a profile, click on the checkbox to load a profile and then click on the '…' button or type the profile name directly in the edit box to the right of the '…' button. Please note that the profile must exist in the profiles subdirectory off the HOTAS directory. To set the startup options back to the default, click on the 'Default' button.

Calibration

The Calibration section contains options to set the calibration mode, calibrate the Joystick, retrieve the auto calibration data and to set the center positions of the axes.

Choosing Auto or Manual calibration in the Calibration Mode option will automatically switch the Joystick to the desired option. Please note that it is not required to click on the 'Apply' button to apply the calibration mode – this happens immediately.

In Manual calibration mode, the data used will be that which was loaded into the Joystick after performing a Manual Calibration. This data is automatically downloaded, along with the current axis parameters, for use after any Calibration Routine is performed.

Auto calibration mode means that the Joystick will configure the axes to their maximum positions as you move the axes to the extremes. Manual calibration means that the Joystick will use the calibration data that was created by performing the manual calibration routine, described later in this section. Please note that the Joystick can not switch to Manual calibration mode if no manual calibration has been performed. If the Joystick is reset (by reconnecting it, or clicking on the Restart button) and is in Auto-calibrate mode, it is recommended that you move all the Cougar axes (joystick, throttle, Range, Antenna, Microstick etc) to their maximum and minimum positions, holding them at these positions for about 3 seconds. This will allow the auto calibration to gather information from the axes and accurately calibrate your controller axes.

Clicking on the "Get Auto Calibration" button before switching the "Calibration Mode" from "Auto calibration" to "Manual calibration" copies the auto calibration data to the current profile. You can now apply and save the current profile so that the Joystick will use this data, and never proceed to adjust its values.

The Set Center Positions button is used to save a specific position of each axis as the center position. If you wish to have the Joystick and the Range axes to have different center positions than the ones given by auto calibration, then you can press this button and specify the desired axes, shown below.

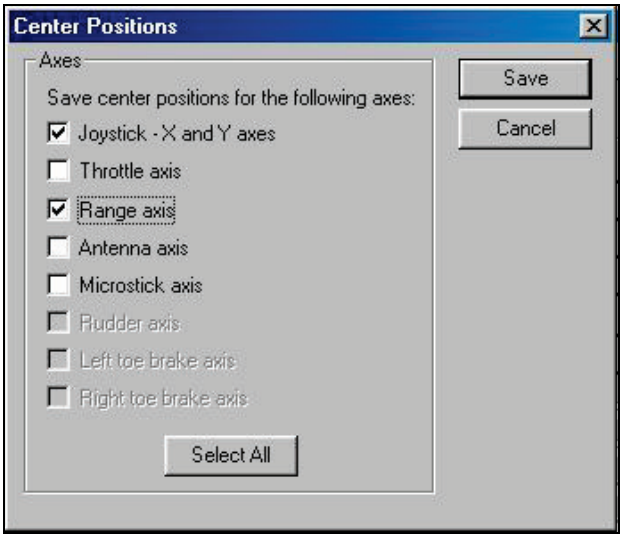


Figure 12: Saving the Joystick and Range Center Position

Once you press the 'Save' button, you will be prompted to move the specified axes to your desired center position and then to press 'OK'. This will now give the specified axes new center positions.

Manual Calibration

The Manual Calibration button will bring up the Calibration Routine window, shown below:

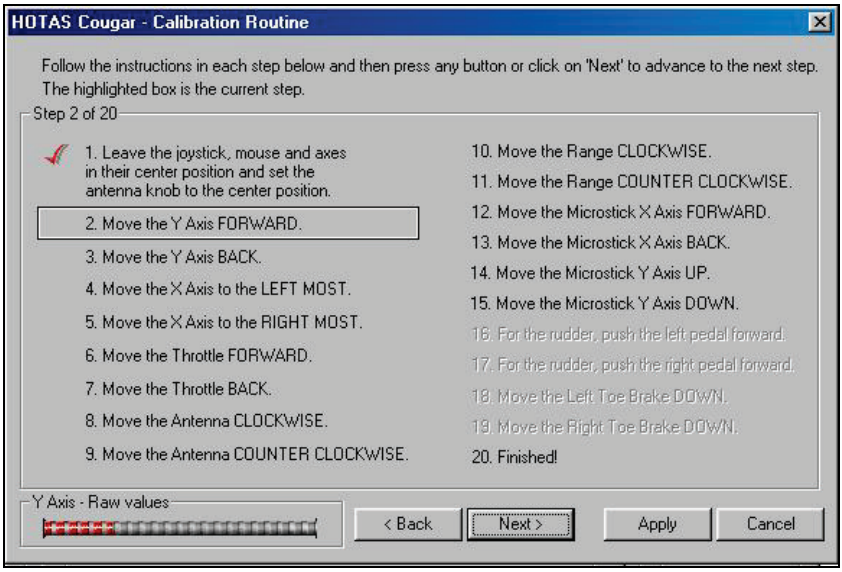


Figure 13: Manual Calibration Routine Window

Follow the directions given, pressing all buttons successively, and clicking on the Next button when the axis has reached the position that is to be saved for that particular step. For instance, if in the second state you move the Y axis to its forward position and then click the Next button, the Joystick will save the position that the Y axis was in at the time when you clicked the button. When you are done with the calibration setup, the application will automatically send the calibration information to the Joystick, along with all the data currently loaded into the axis parameter section. This calibration data is saved, and can be saved to a profile for later use.

Note that the axes that are not physically connected are grayed out. The calibration routine will skip over the axes that are not connected. The progress bar at the bottom left hand side of the Window indicates the 'raw' values of the axis – it may not be possible to reach the maximum or minimum positions of the progress bar. It is to be used as a guide to show the correct axis to be moved and the correct direction.

Actions and other options

In the Actions section there are three buttons: Restart Device, Button & Axis emulation and Download to device. There are also options for the automatic polling of the Joystick and the Hide button. All these actions and options are described below.

RESTART DEVICE

The Restart Device button will cause the Joystick to perform a manual 'unplug', this is similar to disconnecting and reconnecting the Joystick from the USB port. This function is useful whenever you wish to change the Windows Axes States. In this case, it would be necessary to disconnect and reconnect the Joystick (as explained in the section "Changing Windows Axes States"), which is effectively accomplished by clicking on the Restart Device Button. Also, on startup, the Auto Calibration Routine measures the centre positions of the relevant axes (X, Y, Rudder, and microstick axes); to manually set the centers of the axes, click on the Restart button, and hold the axes at their desired positions.

BUTTON & AXIS EMULATION

If the Button & Axis emulation is ON (green background), then the Joystick will use the last emulation file which was downloaded to the Joystick. The emulation file is the file that controls the keyboard and mouse emulation, as well as the various axis parameter changing features. Please refer to the Cougar Owner Reference Book for further information.

If the Button & Axis emulation is OFF (red background), then the Joystick will behave like a simple Joystick with buttons activating DirectX buttons in Windows.

DOWNLOAD TO DEVICE

The Download to device button opens the HOTAS Cougar Loader application. The Loader application is used to download joystick files (Thrustmaster Joystick files .TMJ). For more information about the structure of an emulation file, please see the document "HOTAS Cougar Owner Reference Book." On the next page is a picture of the HOTAS Cougar Loader application.

There are two buttons and a checkbox on the main HOTAS Cougar Loader window. The Check button is to check the joystick file for errors. This will not download the file to the Joystick. The Download button is to check the joystick

file and then download it to the Joystick if there are no errors. The section in

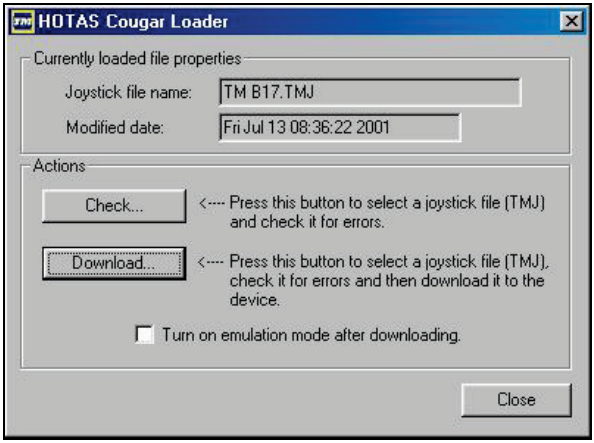


Figure 14: HOTAS Cougar Loader to download joystick files

'Currently loaded file properties' shows you which joystick file is currently loaded in the Joystick and the date the last download was performed. The checkbox for 'Turn on emulation mode after downloading' is used for turning the emulation mode on after a joystick file has been downloaded to the Joystick. Once a joystick file has been downloaded, the emulation mode must be turned on for the Joystick to emulate the keypresses and axis movements stored in the joystick file. This can also be done by clicking the Button & Axis emulation button in the HOTAS CCP. See the section 'Button & Axis emulation' in the last section. When checking or downloading the joystick file, the following window will appear.

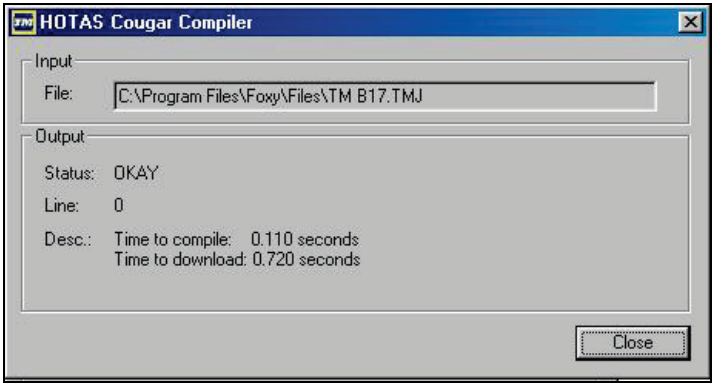


Figure 15: HOTAS Cougar Compiler to check joystick files

POLL DEVICE

The checkbox for polling the device is used for the HOTAS CCP to poll the device for its current status. Polling will check whether the Joystick is connected and which mode the Joystick is in. The interval time for the polling is set in the edit box underneath the checkbox.

HIDE / TASKBAR ICON FUNCTIONALITY

The HOTAS CCP can be hidden with only the icon showing in the taskbar. Click on the 'Hide' button to hide the HOTAS CCP. Note that this functionality can only be used with the polling turned on. To show the HOTAS CCP after the application has been hidden, click on the taskbar icon with the left button and choose 'Open HOTAS Cougar Control Panel…' from the menu that appears. The other options available from the menu are to exit the HOTAS CCP and to open the HOTAS Cougar Loader. You can switch the Joystick to and from emulation mode by clicking on the taskbar icon with the right button. The colors of the icon will change according to the following table:

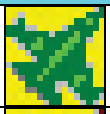
Icon	Colour Code	Description
	*Green fighter / Yellow BG*	Emulation mode is **ON**, Axis response is in **User** mode
	*Red fighter / Yellow BG*	Emulation mode is **OFF**, Axis response is in **User** mode
	*Green fighter / Gray BG*	Emulation mode is **ON**, Axis response is in **Windows** mode
	*Red fighter / Gray BG*	Emulation mode is **OFF**, Axis response is in **Windows** mode

Table 2: Taskbar icon descriptions

THRUSTMASTER

HOTAS COUGAR

OWNER'S GUIDE AND REFERENCE BOOK

CONTENTS

1. What we have in store for you!

1.1 INTRODUCTION

Welcome to the next generation of Thrustmaster high-end controllers - the HOTAS Cougar. Sealed within the hull of this distinctly heavy-metal flight gear package is the controller itself, plus a CD-ROM containing a whole load of utilities and goodies, *plus* this monolithic reference book, and the traditional by now Quick Install guide.

Among the goodies located on the enclosed CR-ROM, you will find all the software material necessary to get your HOTAS Cougar up and running smoothly: the HOTAS Cougar Control Panel (extensively described in the previous section), as well as the Foxy suite of applications, comprising the main programming application, Foxy HOTAS Cougar Edition with all of its components (such as the Composer and Korgy), and its supporting applications, such as Foxy GUI, the Launcher etc.

I will not broach anything here with respect to the HOTAS Cougar Control Panel software – for tons of info on this feature, just keep reading!
Foxy and Foxy GUI, however, are not showcased within this reference book. For all necessary explanations regarding their utilization, please refer to the respective software's online help.
Foxy is your key to easily programming each of your joystick's functionalities, defining powerful and intricate definition and macro files with unequalled facility.
The Foxy GUI will enable you to precisely assign, with just a few mouse clicks, an array of key strokes to a single move of your HOTAS Cougar; you are thus able, with no particular knowledge of the Thrustmaster programming code, to program keyboard functions to the various (and many!) buttons on the stick.

With this said, pop your favorite CD in your hi-fi system, serve yourself a loooong drink, and prepare to spend the forthcoming days (and nights) delving into the promising depths of this reference book!

1.2 SETTING UP YOUR CONTROLLERS

Please refer to the Quick Install provided with the HOTAS Cougar.

1.3 GETTING ACQUAINTED WITH THE REFERENCE BOOK

Obviously, a cutting-edge controller such as the HOTAS Cougar requires the extensive help and support components to match its unequalled potential. So, once you've installed your Cougar and all the software from the CD, and you've checked in the Cougar Control Panel (CCP), as well as Windows' Control Panel's Gaming Options applet that everything seems to be working, you're no doubt wondering where to start with all of this. Hopefully you've installed Foxy as well, and run it (***after*** having run the CCP first though - this is important), and looked at it in horror thinking "Ok, so where do I start with all of this?"

So starting with the real basics, and moving on, here's how you can get going with your Cougar.

Level 1: Basic use

Well you'll be pleased to know that you don't actually need to do anything else, to start using your Cougar. I recommend first that you move all the axes to the ends of their travel, and hold them there for a few seconds to allow the auto-calibration to set itself up correctly. Now you can just exit all the Cougar software if it's running, go into your game, and use your Cougar. If the game allows you to assign functions directly to the hats and buttons, then you can program them from within the game. The game will see the joystick and throttle for what they are and hopefully assign normal flight functions to them, and may even pick up the extra axes like the Range and Antenna knobs on the throttle, and assign functions to them also. And that's it. Just use your Cougar straight away. The Cougar is in what's called Windows or DirectX mode - which is just some terminology that means that the buttons and hats aren't programmed and can be assigned their functions from within the game.

Level 2: Programming the Cougar with pre-supplied files for your games.

The next stage you may want to get into is to set up the Cougar with pre-programmed functions that we've developed and shipped with the Cougar. We've developed files for over 30 different games, and they can be used to set up the Cougar so that you can do a lot more with it than you could do compared to programming them from within your game. These files are on the CD and can be downloaded using the Cougar Control Panel. But there's an easier way, or 2 easier ways actually. They involve using Foxy or FoxyGUI.

Foxy: Goto the Editor's Favourites menu, and click on the Game you're interested in setting up your Cougar for. This will open up two files in Foxy, the left hand one and main one you see is called the joystick file, and the right hand one, the macro file. You don't need to do anything else with them at this stage. Now go to the Download menu, and click the Download menu item. What this will do is to program your Cougar with the files you just opened. That's it! You're ready to go. Your Cougar is set up for your flight sim/game. You can also click on the coloured

Graphical Layout and/or View ReadMe parts of the Apps toolbar (the 2nd one down) and see what the developer of those files has to say about how to use the file for your sim, either graphically and/or through the ReadMe text file.

FoxyGUI: Even easier. Just follow the on screen instructions in FoxyGUI. Just as above, you choose the game you want to play, press the Download button, and then exit FoxyGUI and go and enjoy your game. Again you can view a graphical layout and the ReadMe file by pressing those buttons.

Level 3: Learning how to program the Cougar.

There's a plethora of help we've developed to teach you how to program your Cougar, so here's what's available - just choose the method that best suits you.

1. The next section of the manual, *(2.1 - Understanding the basics of Thrustmaster Programming)* as well as Foxy's help file, will introduce the basics of programming the Cougar, and they're very easy to follow and understand.

2. In Foxy's Wizards menu, try out the Macro wizard followed by the Joystick wizard. Many people learnt how to program their TM controllers from these 2 very simple wizards alone.

3. Again in the same Wizards menu, there are Tutorial files you can click on, which open up files in Foxy that explain and teach the basics and onwards. You can download these files, change them, and see the effect of your changes - a great way to learn programming.

4. FoxyGUI provides an easy way to program your Cougar as well as explaining how your programming would look if you'd done it in Foxy. You will eventually want to migrate to Foxy if you're only using FoxyGUI, because it's more powerful and quicker to use once you've understood the basics.

5. At any time in Foxy, you can press F1 to get help, or highlight a word in your joystick file and press F1 to get help directly on that word. The help file is huge, and is packed with useful information and covers all the detailed explanations in this manual.

6. Introduce yourself to the Composer and Korgy from Foxy's Insert menu … you'll rarely need to dive into the manual with those two components of Foxy.

One thing I cannot stress enough. Programming the Cougar is very very easy. Just put in a little bit of time with the extensive help we've provided, and you'll reap the rewards in no time at all. And once you're used to the text based approach to programming, with all of its advantages, you'll never want to go back to a purely graphical approach so common to simpler controller software.

2. UNDERSTANDING THE BASICS

2.1 UNDERSTANDING THE BASICS OF THRUSTMASTER PROGRAMMING

2.1.1 Introduction

When it comes to programmability, Thrustmaster joysticks and throttles have always set the standards by which other controllers are measured. Unfortunately they've also had a reputation for being difficult to program, probably arising from the fact that the software that shipped with previous controllers was DOS based, and also because people weren't prepared to put in the time doing the homework to understand these controllers. I can assure you that programming these controllers is far easier than learning some of today's complex flight sims.

What I'm going to do here is to start from the basics and assume you've had no experience with Thrustmaster controllers and their programming. It's unfortunate that we use the term "programming" – a term normally associated with software development and complex programming languages. What we're really doing with Thrustmaster programming is just developing files that assign keyboard characters to the buttons on the joystick and throttle.

2.1.2 The concept of HOTAS

Now in any flight sim, you could fly and control all the weapons, cockpit switches etc. totally from the keyboard. We could make this a little more realistic if we added a joystick, and a throttle, and maybe some rudders, collectively called "controllers." Unfortunately you'd still need to keep looking down to press keys on your keyboard. But what if we put hats and buttons onto a joystick, which when pressed, had the same result as you pressing a keyboard key? Then you'd never have to take your hands off your joystick and throttle, you'd never need to touch the keyboard, and you'd be able to concentrate on flying, weapons control, etc. This is the HOTAS concept, a trademark of Thrustmaster, allowing you to maintain your **H**ands **O**n **T**hrottle **A**nd **S**tick at all times.

2.1.3 So how do we achieve HOTAS for our flight sims and other games?

Quite easily - we program our controllers to mimic the keyboard and the keys that you press in your flight sim. We do this via two files, the **joystick file** which determines which buttons and hats you want to assign keyboard characters to, and the macro file, which contains *"macros"*, which are quite simply descriptions as to what the keyboard characters do in your particular flight sim. Let's begin then by introducing these two files to you.

2.1.4 Introducing the joystick file – the basics of programming

We said earlier that the joystick file is used to assign keyboard characters to the various buttons and hats on your controllers. Now, I guess it is a bit of a misnomer to call it a joystick file. It doesn't just program the joystick. The joystick file is used to program **all** of your controllers - ie. your joystick, throttle and rudders if you have them.

The buttons and hats on your joystick and throttle have got special names, to differentiate them for when we want to program them. Now I'm not going to scare you straight away with a list of all of them. In fact, you never need to learn them really because Foxy, with its Composer, will teach you them as you go along. But I will introduce a few of their names now, as I introduce you to the basics of programming them. Here goes then!

Let's say that we wanted to program button S2 on the joystick to operate the autopilot. Now you can see which is button S2 from the diagram above. It's the top red button on the face of the joystick, just to the left of a large white hat, called Hat 1. Coming back to button S2 then, we want to program button S2 on the joystick to operate the autopilot. Let us also say that normally you'd have to press the "a" key on your keyboard to activate and deactivate the autopilot - that's fairly common in flight sims.

So what we need to do is to tell the joystick to produce an "a" character every time button S2 is pressed. Now, in a joystick file, which is just a simple text file, buttons are identified by the term "BTN" and in this case, we're referring to button S2. We want BTN S2 to send an "a" character. So here's the statement we need to type into our joystick file, to program button S2 to generate an "a" character when it is pressed:

```
BTN S2 a
```

Easy huh! Let's program one of the hats when it is pushed into its up position, to emulate the function key F1 on your keyboard. It might be that in your flight sim, this is the "look forward" key. Now all hats are effectively buttons, so just as before, we can program it with a button (BTN) statement. HAT 1 Up is shortened to H1U so the statement we're after is:

```
BTN H1U F1
```

These statements then are the basics of Thrustmaster programming.

Now in today's modern flight sims, there can be in excess of 100 combinations of keys that make up your flight controls. Trying to remember what they all do becomes a bit of a nightmare. Well we could add a Remark (REM) statement which is a little reminder as to what the button is supposed to do. Like this:

```
BTN S2 a   REM This turns the Autopilot on and off
BTN H1U F1    REM Show the forward view
```

But there's a better way, using something called macros and the macro file. Before we move onto these, it's worth noting that REM statements can be placed anywhere in a file, and that anything after a REM statement on that line is ignored by the joystick. People often use them at the beginning of their files for titles, descriptions, general comments etc.

Let's move on then … "To infinity and beyond!" Well … onwards to the macro file at least!

2.1.5 Introducing macros and the macro file – the basics of programming

Before we come onto a macro file, let's define what a macro is. In the previous section we talked about the joystick file, and we came up with two statements:

```
BTN S2 a  REM This turns the Autopilot on and off
BTN H1U F1    REM Show the forward view
```

Now, a macro is a word that we make up to make it easier for us to remember what a keyboard key or group of them actually does in our flight sim. Like this:

```
Autopilot = a
Forward_view = F1
```

And now we can change our joystick file statements to:

```
BTN S2 Autopilot
BTN H1U Forward_view
```

It may not be obvious as to why this method is better, but believe me, when you've got a joystick file that contains 100+ statements in it, it makes it very much easier to go through and understand it this way.

So where do we put these macro statements?

Well they go into their own file, the macro file. So a macro file contains all the macros that describe what keyboard characters do in your sim, and the joystick file assigns these macros to the buttons on your joystick and throttle. You will therefore normally see Thrustmaster files in pairs, the joystick and macro file, for a particular flight sim or game. This is the reason why in Foxy, the main screen you use is the Editor, which shows the Joystick file and the Macro file on separate tabs. So in our example, let's say it's for the flight sim Falcon 4, we could save the joystick file as Falcon 4.tmj and the macro file as Falcon 4.tmm (tmj = **TM J**oystick File, tmm = **TM M**acrofile.)

2.1.6 How does the joystick file know which macro file contains its macros?

Right, well we've covered the basics of producing joystick and macro files, and hopefully you can see that they're not difficult to understand or produce. Now if like me you have 30 or so flight sims, then that means you've potentially got 30 joystick and 30 macro files around in Foxy's Files folder.

So as the title to this section asked, "How does the joystick file know which macro file contains its macros?" Well, at the moment, it doesn't. We need to tell it which macro file to use. So to complete our basic joystick file, what we need to do now is to tell the joystick file, which macro file contains the Macro DEFinitions (MDEF) it is using. After all in one macro file, wheelbrakes may be activated with a "w" character, and in another it might be the "b" character. We do this with a statement **in the joystick file**, which identifies which macro file to use with this joystick file, like this:

```
USE MDEF Falcon 4.tmm
```

So Foxy reads the USE MDEF line in the joystick file when it opens the file, finds the corresponding macro file, in this example Falcon 4.tmm, and uses the macros from it for programming the joystick file.

2.1.7 Summarising what we've learnt so far ...

Let's take at look then at how our joystick and macro files are taking shape.

Joystick file *(Falcon 4.tmj)*	**Macro file** *(Falcon 4.tmm)*
REM ---------- REM Falcon 4.tmj REM Falcon 4 joystick file REM REM Rem statements don't do anything. REM We use them to add comments REM ---------- REM We tell the joystick file which REM macro file contains its macros USE MDEF Falcon 4.tmm REM Now we program some buttons by REM assigning macros onto them REM from the macro file BTN S2 Autopilot BTN H1U Forward_view	REM ---------- REM Falcon 4.tmm REM REM Falcon 4 macro file REM ---------- REM Macros make it much easier for REM us to remember what actions REM keyboard keys perform in our REM flight sim. REM Macro definitions start here Autopilot = a Forward_view = F1

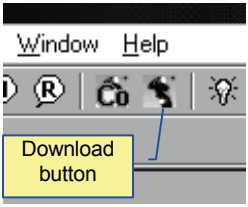
Yes, I know that these macros aren't correct for Falcon 4 - I'm just trying to illustrate a point here!

Now then, we have learnt that:

1. Macro files contain macros, which simply describe what a keyboard key does in your flight sim.
2. Joystick files assign the macros from their associated macro file, onto your joystick and throttle's hats and buttons, via the BTN statement.
3. The joystick file knows which macro file to get its macros from, via the USE MDEF statement.
4. REM statements are simply a way of adding comments to files.
5. Joystick and macro files are quite simply text files, with the extensions .tmj and .tmm respectively, that exist in the same folder on your hard drive. By default this is Foxy's Files folder.

2.1.8 Downloading the joystick file into our controllers

Great! We've covered the basics of developing files for your controllers. So that just leaves us with the burning question, "How do we get our files into our controllers?"



Well the simplest way is to press the "Download" button on Foxy's toolbar, or you could just press "F12" on your keyboard. After a very short period of time, the file will have been transferred, or to use the correct term, "downloaded" to your controllers. And that's it - you're ready to go off flying with your buttons now programmed as laid out in your joystick file. Simple! Before I leave this here, let me just expand a little as to what happens when you download a joystick file to your controllers.

What happens is this: The joystick file is sent to one of the Thrustmaster Cougar applications, called the Compiler. It has the job of converting the joystick file, in combination with the macro file, from a text file into a format that the controllers will understand. This conversion is called "Compiling" and when it is happy that the compiled file has no errors in it, it downloads the compiled information to the controllers. It then sends a message back to Foxy to say that all has gone well, that it has now put the controllers into a state where the buttons will generate the programmed characters when pressed, and with its job done, it exits and passes control back to Foxy.

2.1.9 Structure of joystick and macro files

Before we leave this introductory section and proceed further, I'd like to have a look at some general rules for structuring your joystick and macro files. The examples below demonstrate this. Do **not** at this stage worry about understanding what each line does. All I'd like you to do is just to try and get an idea as to what goes into a joystick file, and what goes into a macro file, and how they are laid out. I'll also draw attention to the fact that a joystick file has a section in it for configuration statements, which we'll discuss later.

Sections	Joystick file *(Falcon 4.tmj)*	Macro file *(Falcon 4.tmm)*
Title Not compulsory, but a good idea.	Rem ---------- Rem Falcon 4.tmj Rem Rem Falcon 4 joystick file Rem Rem last modified 1st Jan 01 Rem Rem ----------	Rem ---------- Rem Falcon 4.tmm Rem Rem Falcon 4 macro file Rem Rem last modified 1st Jan 01 Rem Rem ----------
Configuration statements *(**only** in the joystick file)*	Rem Rem Configuration statements Rem USE MDEF Falcon 4 USE RATE (60) USE TG1 AS DX1 USE S2 AS DX2	Rem Rem Configuration statements Rem Rem don't go into macro files Rem Rem So macro definitions start here

(Continued on following page!)

Command syntax Joystick file Button assignments, Axes statements, Logical programming. Macro file Macro definitions	Rem ---------- Rem Button assignments Rem ---------- BTN H1U View_up BTN H1D View_Down BTN H1L View_Left BTN H1R View_Right BTN S1 Cycle_MSL_hardpt BTN S2 Pickle_weapon BTN S3 /U Cycle_RDRsubmode /M Ground_Map_FOV /D Cycle_RDRsubmode BTN S4 /T Padlock_view /T 2-D_cockpit Rem ---------- Rem Throttle Rem ---------- BTN T2 /T Virtual_Cockpit /T 2-D_cockpit BTN T3 Look_Closer BTN T4 Padlock_Next BTN T5 Padlock_Prev BTN T6 Uncage	Rem ---------- Rem View control Rem ---------- View_up = KP8 View_Down = KP2 View_Left = KP4 View_Right = KP6 Rem ---------- Rem Weapons Rem ---------- Cycle_MSL_hardpt = SHF / Pickle_weapon = SPC Rem ---------- Rem Miscellaneous Rem ---------- Cycle_RDRsubmode = F8 Ground_Map_FOV = F9 Padlock_view = 4 2-D_cockpit = 2 Virtual_Cockpit = 3 Look_Closer = l Padlock_Next = KP+ Padlock_Prev = KP- Uncage = u

That then is a quick introduction to the basics of programming. Now it's time to look at the statements that make up a joystick file in detail. We'll begin with the one we introduced initially – the button (BTN) statement, identify what all the buttons and hats are called, and discuss macros in a little more depth.

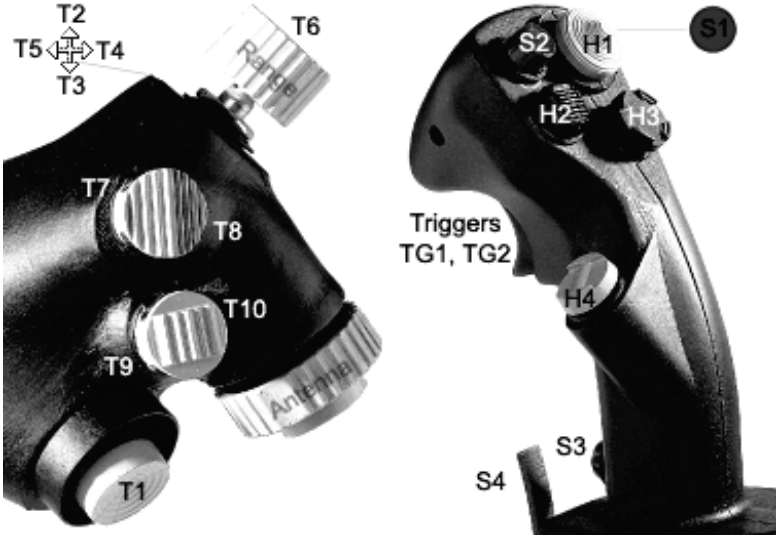
3. Button statements and Macros

3.1 BUTTON STATEMENTS AND TM KEY SYNTAX

The HOTAS Cougar consists essentially of several axes, a number of hats, buttons, trigger etc. Anything that isn't an axis is programmable through a button statement with the syntax:

Command syntax
BTN Button_name KeySequence and/or macro/s

where:
Button_name identifies the button to be programmed:

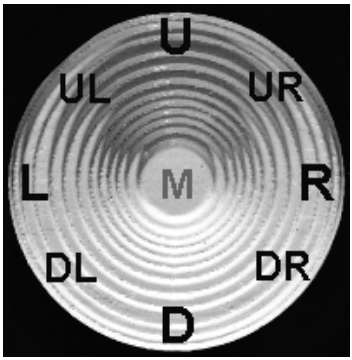


The Throttle has:	10 buttons:	T1 to T10
The Joystick has:	4 hats :	H1 to H4
	4 switches:	S1 to S4
	2 stage trigger:	TG1, TG2

Examples:

```
BTN T3 y              Rem "Roger, understood old fruit"
BTN S2 Eject
BTN T4 Chaff Flare    Rem Time for bowel movements
BTN S4 h e l l o      Rem Notice the spaces – it's not a macro
```

Each hat has 9 programmable positions:



In general only the 4 main positions are programmed. For HAT 1 for example these would be:

```
BTN H1U  Look_up
BTN H1R  Look_right
BTN H1D  Look_down
BTN H1L  Look_left
```

But the corner positions are also programmable:
e.g. BTN H1UL View_UL

and the middle position:

```
BTN H1M  View_forward
```

NOTES

1. *The BTN statement does not need to be right at the beginning of a line, but only one BTN statement is allowed per line, and you cannot reference the button more than once in a file. So if you have:*

```
BTN S2 a b c
BTN S3 d e f
```

then these statements are fine, but if you then further in your file have:

```
BTN S3 g h I
```

then the compiler will generate an error informing you that you've got a duplicate button statement.

2. *There must be a single space between BTN and its Button_name, so:*

```
BTNS2
```

will generate an error.

3. *You must also have a space after the BTN Button_name, so:*

BTN S2a b c

will generate a compiler error.

4. *A character or macro is produced only once with a button statement, irrespective of whether you hold down the button or not. If you want it to repeat, then you can consider either using the /A auto-repeat slash modifier, or the /H hold modifier. This is a different behaviour to the original TM syntax, and is by design. Slash modifiers are covered later in this reference book.*

ADVANCED NOTES

I'm going to digress slightly here and discuss the hat's middle position. I don't suggest that you try to take this in now, but just be aware of what's to follow just in case it arises in one of your files one day. The centre of each hat can be programmed by adding the M to the hat, as in the example above. Note that if you have a /P, /R (see later notes on slash modifiers) programmed to one of the hat positions, the /R keys will be generated at the same time as the M keys. So:

```
BTN H1U     /P 1
            /R 2
BTN H1M a
```

will generate, when HAT 1 is pressed up and then released:
***"1", then "a" and "2"** at the same time.*

If you wanted to ensure that the H1M statement executed after the H1U /R statement, then you could add a delay (see later notes) on the H1M statement like this:

BTN H1M DLY(60) a

3.2 THRUSTMASTER KEYBOARD SYNTAX

Coming back to our statement syntax:

Command syntax
BTN Button_name KeySequence and/or macro/s

Let's take a look at this last part – key sequence.

If we're going to assign keyboard characters, either in macros or in joystick statements, you need to understand that there is a Thrustmaster style of identifying each keyboard character, which we call Thrustmaster Key Syntax. The basis behind this is that for example, pressing the "5" key on your main keyboard may be totally different in your simulator to pressing the "5" on your numerical keypad on the right of your keyboard. So we need to be able to distinguish them, with a '5' and a 'KP5' in this example. This then is the Thrustmaster Syntax for keyboard keys:

ESC	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12		
`	1	2	3	4	5	6	7	8	9	0	-	=	BSP	
TAB	q	w	e	r	t	y	u	i	o	p	[	]	\	
CAPS	a	s	d	f	g	h	j	k	l	;	'	ENT		
LSHF	z	x	c	v	b	n	m	,	.	/	RSHF			
LCTL	LALT	SPC								RALT	RCTL			

PRNTSCRN	SCRLCK	BRK

INS	HOME	PGUP
DEL	END	PGDN

	UARROW	
LARROW	DARROW	RARROW

NUML	KP/	KP*	KP-
KP7	KP8	KP9	KP+
KP4	KP5	KP6	
KP1	KP2	KP3	KPENT
KP0		KP.	

The easiest way to ensure that you get these correct is to use Korgy, the virtual keyboard in Foxy. You can also display this information from Foxy's Editor from its Help menu - by selecting the "Keyboard Syntax" menu item.

NOTES

1. *The Syntax for chorded keys (where you hold down the Shift, ALT or CTRL keys) is* SHF a, ALT b, CTL c. *These are not the same as using* LSHF a, LALT b or LCTL c. *For example,* LSHF *a is equivalent to pressing the left shift key on your keyboard, then releasing it, and then pressing the "a" key, and then releasing it. By default the compiler uses the* ***left*** *Shift, ALT, CTRL keys for chorded keys.*

2. *Some keys are reserved:* ( ) { } < > *and need to be programmed with* SHF *statements:*

(= SHF 9
) = SHF 0
{ = SHF [
} = SHF]
< = SHF ,
> = SHF .

3. *A good habit to get into is to always use* SHF, ALT *or* CTL *in conjunction with other keys, instead of capitalising them:*

BTN S1 SHF p
BTN S1 P

Both are correct, and produce a "P" but the former is a better habit to get into.

4. *The keyboard layout and syntax is based on a US keyboard. There are sometimes situations where you may need to produce a keyboard character that exists only on a different keyboard layout. If this is the case, you can do this using USB codes directly … this is discussed later in the reference book and will be likely be used very rarely. But you never know, so it's there!* ☺

5. *The syntax has changed from the original TM F22 and is by design. So for example the AUX suffix on some keys has been removed.*

3.3 MACROS AND MACRO RULES

In the Introduction section, we introduced the concept of macros and the macro file. I gave two examples, which were:

Autopilot = a
Forward_view = F1

Now that we've covered the syntax for all the keys on the keyboard, then you're in a position to be able to generate a macro file containing macros for your flight sim or game. Foxy has several useful utilities to help you generate macros that will ensure that you create macros with the correct syntax, and that follow the rules below. These are: The Macro Wizard, Speedy and Korgy. *See the Foxy documentation on these for further information*. Before I get onto the macro rules and regulations, let me throw in a quick tip here. It can be very boring and arduous to sit with your game's keyboard layout card, creating all the macros. If the game has a help file, see whether you can copy all the keyboard assignments from that, and then just convert that text into macros that adhere to the rules below. One final point, I regularly receive files from people who are having problems successfully downloading files to their controllers, because the Compiler generates errors that aren't immediately obvious. I have learnt from

experience that whenever I get sent files, I always run through the macro file first and look for incorrect syntax, or macros that break the rules below. And I would say that 90% of compiler errors arise through mistakes in the macro file. Take time then to create your own macro files, because you know then that they are correct and once created, you can have the time of your life setting up the joystick file!

Now to the rules and regulations …

1. Macro names **cannot** contain spaces. Use an underscore _ or - instead. So:

My macro is = b

is not allowed whereas:

My_macro_is = b
My-macro-is = b *are.*

2. Make sure that you have a space before **and** after the "=" sign following the macro name. So both of these macro definitions:

Autopilot= a
Autopilot =a *are incorrect.*

3. You **cannot** use the following characters in macro names:

= < > { } ( ) ^ , *spaces*

4. Try not to use capitals for macro names. eg. RADAR_RANGE_INCREASE. Although you can, a file is much easier to read if you don't. It is also good syntax to reserve capitals for TM commands (eg. MDEF), or abbreviations (eg. HUD) only.

5. Macro names are case insensitive. So:

MyMacro = a
mymacro = a are the same.

ADVANCED NOTES

I'll throw this one in here for the real hardcore amongst you!

Macros can be nested, ie. you can use a macro to call another macro. Like this:

Macro_1 = a b c
Macro_2 = Macro_1 d e f

You can have up to 20 nested macros for one macro, i.e.

Macro-1 = a Macro-2
Macro-2 = b Macro-3
Macro-3 = c Macro-4
... etc...
Macro-20 = d

But not more than 20. And once you get this...

Macro-1 = a Macro-1

The compiler will generate an error stating: "Macro looping too long; two macros might be calling themselves."

3.4 STATEMENT MODIFIERS

We've looked at some simple statements such as these:

BTN T4 a

which results in a single "a" key being generated when Button T4 on the throttle is pressed and released. Remember though that there will be situations when you don't want just a single character produced. On a keyboard, you can press keys to generate a string of characters, with delays in between each key, you can hold down a single key, or a group of them, etc. **The basis for any successful programmable controller is that it must be able to emulate what you can do with a keyboard**.

And this is where we introduce Statement Modifiers. We want to be able to go beyond producing single characters from a button.

Statement modifiers are statements that can be used to change the behaviour of characters programmed onto a button. They come in 5 flavours:

1. Slash modifiers /U, /M, /D, /I, /O, /P, /R, /T, /A, /H

```
BTN S4 /H b   Rem Wheelbrakes
BTN T3 /A c f  Rem Chaff and flares
```

2. Delay and Repeat statements DLY( ), RPT( )

```
BTN T6 1 DLY(60) 1 DLY (60) 2 Rem Request Vector For Recovery TAW
BTN S2 RPT(6) c  Rem 6 chaffs please … like right now would be good!
```

3. Character grouping – using brackets ( ), { }, < >

```
BTN T2 (a b c)
BTN T3 {a b c}
BTN T4 /P <a b c>
       /R d
```

4. Working with and defining DirectX (Direct Input) buttons DX

```
USE TG1 AS DX1
BTN H2U DX1
USE ALL_DIRECTX_BUTTONS
```

5. Using KeyDown, KeyUp and USB codes KD( ), KU( ), USB( )

```
BTN H4U KD(a) DLY(60) KU(a)
BTN H4D /P USB (D51) /R USB (U51) Rem 'Down arrow'
```

We'll take a look at slash modifiers to begin with …

6. *You cannot use reserved TM syntax in your macro names. This includes the TM syntax for the keyboard keys, and other syntax used in the various TM statements. So if you have in your joystick file:*

```
BTN S2 HOME
```

and in your macro file:

```
HOME = k
```

then the compiler will generate an error if you try to compile or download that joystick file, because the word "HOME" is the TM syntax for the keyboard HOME key.

3.5 SLASH MODIFIERS

There are 10 slash modifiers in total, and they can be grouped together into 3 main groups of slash categories, based on the effect they have on buttons and their statements:

Increasing the number of programmable positions:

/U, /M, /D using the Throttle's Dogfight switch (T7, T8)
/I, /O using the Joystick's Button S3 switch

Separating out macros on a button:

/T Toggle
/P, /R Press and Release

Repeating and non-repeating characters:

/A Auto-repeat
/H Hold

Slash code hierarchy and rules:

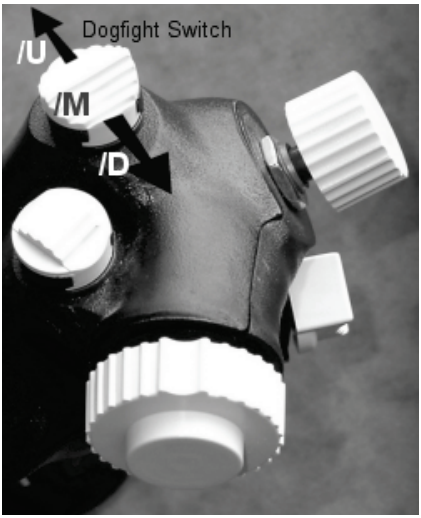
Finally, you'll need to understand where and where you can't use slash modifiers, and in what order.

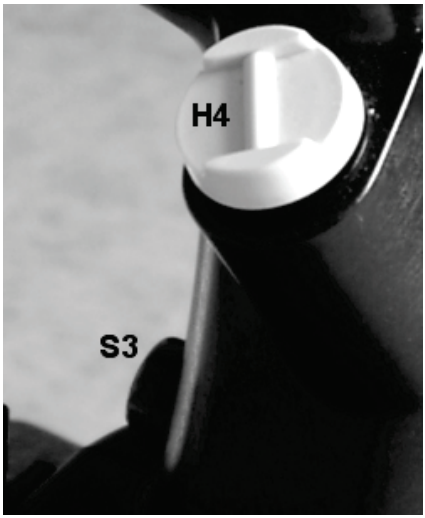
3.5.1 Increasing the number of programmable positions:

The Dogfight switch on the throttle, can be moved into 3 different positions - Up (/U), Middle (/M), Down (/D). Similarly, button S3 can be pressed in (/I) or be out (/O), *see diagrams below*. You can use a combination of these to increase the number of programmable positions of a button/hat/axis by up to 6 times.

Statement modifiers

/U, /M, /D using the Throttle's Dogfight switch (T7, T8)
/I, /O using the Joystick's Button S3 switch





3.5.1.1 /U, /M, /D - Up, Middle, Down

You can increase the number of programmable positions on a button , HAT or digital axis using the position of the Dogfight switch, with the /U, /M and /D slash modifiers. For example:

```
BTN H4U /U a
        /M b
        /D c
```

When Hat 4 is pressed up, it will produce:

- an "a" character if the dogfight switch is in its up (/U) position
- a "b" character if the dogfight switch is in its middle (/M) position
- a "c" character if the dogfight switch is in its down (/D) position

You **must** place /U, /M, /D statements on separate lines. If they appear on the same line then a compiler error will be generated. They must also appear in that order. So:

```
BTN H4U  /D a
         /M b
         /U c
```

will generate a compiler error.

3.5.1.2 /I, /O - In, Out

You can increase the number of programmable positions on a button , HAT or digital axis using Button S3, with the /I and /O slash modifiers. For example:

```
BTN H4D  /I 1
         /O 2
```

When Hat 4 is pressed down, it will produce:

- a "1" character if button S3 is pressed **I**n (/I)
- a "2" character if button S3 isn't pressed (/O = **O**ut)

Combining /U, /M, /D with /I, /O slash modifiers in statements

You can also combine these slash modifiers. For example:

```
BTN H4R    /U /I   Engage_my_target
           /O      Break_right
```

```
/M /I   Camera_right
  /O    Next_waypoint
/D /I   Engine_right
  /O    View_right
```

So now Hat 4 when pressed to the right will perform 6 different functions, depending on the position of the Dogfight switch (/U, /M, /D) and button S3 (/I, /O).

NOTES

1. *The respective* /U, /M, /D *slash modifiers, if present always precede the* /I, /O *modifiers.*

2. *You cannot use* /U, /M, /D *slash modifiers on the throttle's dogfight switch (T7 and T8 buttons), as they're obviously used to determine these positions.*

3. *You* ***must*** *place* /I, /O *statements on separate lines. This is different to the original TM HOTAS syntax.*

4. *You must have the* /I *statements* ***before*** *the* /O *statements. So:*

```
BTN H4D   /I 1
          /O 2
```

is a valid statement but these two statements:

```
BTN H4D   /O 2
          /I 1
BTN H4D   /I 1 /O 2
```

will both generate compiler errors. This is different to the original TM HOTAS syntax.

5. *If you use* /I, /O *modifiers on S3, any statements on its* /O *position will usually be ignored by the Compiler. I say usually because if you use* S3_LOCK *statements (see later notes), then you can generate statements on the* /O *position of the S3. If you define a different button to use for S3, using the* USE *Btn* AS SHIFTBTN *statement (see later notes) then the same notes apply.*

6. *If a file has been written for a joystick and a throttle, but only the joystick is present, then the compiler will use the* /M *codes on any button/hat position, and* /U, /D *statements will be ignored.*

ADVANCED NOTES - HOW STUCK KEYS ARE PREVENTED

Some technical notes for you as to what happens when you change the S3 state (i.e. going from pressed to not pressed, or vice versa) while a button which is programmed with both /I and /O statements is being held. If the dogfight switch's position changes, or the joystick's S3 state changes, this is what the controllers do:

1. *Cougar recognises change of state*
2. *Cougar checks for held buttons*
3. *Cougar checks held buttons to see if they are programmed differently in the new state.*
4. *If so, adds the "release" Macro from the previous state to the output. No key-presses are performed, but all others are (eg. Mouse statements, DirectX statements, axis statements)*
5. *Cougar is in new state.*

3.5.2 Separating out macros on a button:

Statement modifiers

/T Toggle between different macros on a button
/P, /R Press and Release actions on a button

3.5.2.1 /T - Toggle Slash modifier

The easiest way to understand what a /T toggle modifier does is just to look at an example, and see what effect it has:

```
BTN S2 /T a
       /T b
       /T c
```

Let us say that button S2 is pressed 3 times. On the first press, an "a" character is generated. On the next press, a "b" character, and on the final press, a "c" character. If it is pressed again, then an "a" character is produced, and the cycle repeats, as the button toggles through its characters/macros.
There are 16 possible toggles per programmable position, including /U, /M, /D, /I and /O statements, i.e. you can have:

```
BTN S2 /U /I   16 toggles can go here
          /O   16 toggles can go here
       /M /I   16 toggles can go here
          /O   16 toggles can go here
       /D /I   16 toggles can go here
          /O   16 toggles can go here
```

NOTES

1. *Toggle slash codes are* ***not*** *permitted after* /P *or* /R *statements. So if you have:*

Example 1.
```
BTN TG1 /T /P a
        /R b
    /T c
```

Example 2.
```
BTN TG1 /P /T a /T b
        /R c
```
Example 1 is permitted, but 2 and 3 will generate compiler errors.

Example 3.
```
BTN TG1 /P a
        /R /T b /T c
```

2. *You cannot use* /T *statements on T1 if you have a* USE T1_SENSITIVITY *statement in your joystick file (see later notes).*

3. *You cannot use* /T *statements on a joystick Hat's statements if you have a* USE *HatID*_SENSITIVITY *statement in your joystick file (see later notes).*

4. *You cannot use* /T *statements with Digital axes statements (see later notes).*

5. *You cannot use* /T *with logical programming statements (see later notes).*

6. *A single* /T *modifier in a statement will generate a compiler error. So:*

```
BTN T4 /T a
```

is not permitted, as there must be more than one /T *in a statement.*

7. /T *statements can appear on the same line or different lines. So:*

```
BTN S2 /T a /T b /T c
```
is permitted

8. *You can reset the toggle order with the* RESET_TOGGLES *statement, and you can reverse the direction of toggling with the* REVERSE_TOGGLES *statement, both of which we're coming onto.*

9. *You can use other slash modifiers with* /T *that you couldn't with the original TM controllers. For example:*

```
BTN S2 /T a /T /H b
```
is permitted.

10. *You cannot use toggles on the middle position of a hat. So this will generate a compiler error:*

```
BTN H1M /T a /T b
```

3.5.2.2 Resetting the toggle position

If you want to reset the toggle index so that you're back at the beginning of a toggle statement, you can do so easily with:

Command Syntax

RESET_TOGGLES

So if you have a statement such as:

```
BTN S2 /I RESET_TOGGLES
       /O /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
```

and you've pressed button S2 several times so that you are generating a '7' character for example, then you may want to quickly get back to the beginning of the series of the toggles. You can do this pressing button S2 with button S3 (which activates the /I statement) held in.
Pressing button S2 whilst S3 is in won't generate any characters. You'll need to press S2 with button S3 not pressed, which would generate a '1' character with the above statement.

NOTES

1. *This statement **must** appear directly after a /I or /O statement, with the toggles appearing on the corresponding /I or /O statement. So this would generate a compiler error:*

```
BTN S4   /U RESET_TOGGLES
         /M /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
         /D Some_macro
```

but this would be fine:

```
BTN S4   /U Some_macro
         /M /I RESET_TOGGLES
            /O /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
         /D Some_macro
```

2. *A RESET_TOGGLES statement cannot have anything else on its statement line.*

3.5.2.3 Reversing the direction of toggling

If you want to reverse the direction of toggling on button, then you can do this with:

Command Syntax

REVERSE_TOGGLES

```
BTN S2 /I REVERSE_TOGGLES
        /O /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
```

Normally pressing button S2 will toggle between the 1 to 0 characters. If you press button S2 now with button S3 held in, then you will step through the toggles in the reverse direction from the toggle position you were in.

NOTES

1. *This statement **must** appear directly after a /I or /O statement, with the toggles appearing on the corresponding /I or /O statement. So this would generate a compiler error:*

```
BTN S4 /U REVERSE_TOGGLES
        /M /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
        /D Some_macro
```

but this would be fine:

```
BTN S4 /U Some_macro
        /M /I /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
           /O REVERSE_TOGGLES
        /D Some_macro
```

2. *A REVERSE_TOGGLES statement cannot have anything else on its statement line. So if we take the above example and change it to:*

```
BTN S4 /U Some_macro
        /M /I /T 1 /T 2 /T 3 /T 4 /T 5 /T 6 /T 7 /T 8 /T 9 /T 0
           /O REVERSE_TOGGLES Macro_here_generates_error
        /D Some_macro
```

then the macro after the REVERSE_TOGGLES statement will generate a compiler error.

3.5.2.4 /P, /R - Press and Release

The /P slash modifier indicates that the specified macro which follows it is to be generated upon the press of the button or switch. The /R slash modifier indicates that the specified macro which follows it is to be generated upon the release of the button or switch.

For example: BTN S2 /P Chaff
/R Flare

In this example, BTN S2 is using the press and release modifiers. Button S2 will generate the Chaff macro when pressed, and the Flare macro when released.

NOTES

The /R modifier must be used in conjunction with the /P modifier. If no /P is present, then using a /R will generate a compiler error, and vice versa.

ADVANCED NOTES

1. *If you use these modifiers with a hat position, in conjunction with a statement for that hat's middle position, then /R macro's characters will be generated at the same time as the BTN HxM statement's characters. So:*

```
BTN H1U /P 1 /R 2
BTN H1M a
```

will generate when HAT 1 is pressed up and then released:
*"1" and then "**a 2**" together*

2. *If you use these modifiers with the hat middle positions BTN HxM then moving the hat away from the central position will generate the keys associated with the HxM /R statement at the same time as those associated with H1U. So:*

```
BTN H1U 1
BTN H1M /P a /R b
```

will generate when HAT 1 is pressed up and then released:
*"**b 1**" at the same time and then "a"*

3. *There is the potential to produce sticky (continually repeating) keys when using /P /R statements. For example, consider this:*

```
BTN S4 /P DLY(2000) KD (p)
       /R KU (p)
```

Pressing button S4 will first cause a delay of 2 seconds, then emulates the "p" key on your keyboard being pressed down, and on the release of the button will act like taking your finger off the "p" key.

But what happens if you take your finger off button S4 before the first delay of 2 seconds is over?
*Although the delay continues, and the press of the "p" key will occur after that 2 second delay, the action of the release of the button S4 and thereby the release of the "p" key will be processed before that press. Unfortunately, the controller sent a "get your finger off the 'p' key" command before you even sent the command "put your finger on the 'p' key" command, and so the "p" key will appear to be stuck. There are 2 ways to avoid this. The first is to use and program your HOTAS Cougar sensibly. If you know that you have to keep your finger on a button for a certain length of time, then do so! That will **always** avoid any sticky key problem. The other way is to enclose the /P statement within < > brackets:*

```
BTN S4 /P < DLY(2000) KD (p) >
       /R KU (p)
```

which forces everything within those brackets to be completed before the /R statement can be conducted, even if you take your finger off the button early. Note that all macros on all buttons if not currently executing get stalled until the < > statement is completed. So be careful when using it!

3.5.3 Repeating and non-repeating characters:

Statement modifiers

/A Auto-repeat
/H Hold

3.5.3.1 Non repeating characters

The behaviour of macros and single characters on a button statement has changed since the original TM syntax. By default, every button and digital statement, whether there are single characters on it, or macros, do **not** repeat. Just a single instance of the character/macro etc. is produced. For those who are used to the original TM HOTAS, it's as though there's an invisible /N slash modifier in front of every statement if no other is there.

Therefore, a /N modifier is no longer needed - in fact it is ignored by the compiler, if one is seen in a file. I recommend that you delete them from your files if present.

3.5.3.2 /A - Auto-Repeat

This slash modifier results in the exact opposite functionality. It forces the characters/macros on a button or digital statement to continually generate until the button is released.

```
BTN S2 /A Fire_Missiles
```

Now, pressing button S2 will force the Fire_Missiles macro to repeat. It's as though you are rapidly pressing **and** releasing a key on your keyboard. If other buttons are then pressed on the controllers, their output will feed into the stream of characters being generated by this statement. ie. pressing another button does not stop characters being generated by this statement. Note macros following a /A modifier can consist of many characters, DLY, RPT etc statements. Also, the /A modifier replaces the previous TM syntax of placing brackets around a key/macro to force it to repeat.

3.5.3.3 /H - Hold

```
BTN S4 /H b  Rem Wheelbrakes
```

A /H Hold statement can be likened to holding down a key on your keyboard (*although see the Notes section below*). For example, many flight sims require you to hold down the "b" key on your keyboard for controlling your wheelbrakes. As soon as you release the key, the wheelbrakes come off.

Some more examples:

```
BTN S4 /H b  Rem Wheelbrakes
BTN T6 /H Wheelbrakes
BTN T1 /H ALT F6
```

It is also possible to use the /H modifier in more complex statements. In this example below, the statement is equivalent to producing a single "c" character, followed by a held down "f" character.:

```
BTN S4 /H c f Rem 1 Chaff, loads of Flares
```

Whereas:

```
BTN S4 /H {c f} Rem Chaff and Flares
```

produces KD(c) KD(f) until the button is released, when it produces KU(c) KU(f). (*See later notes on KD and KU statements.*)

An interesting point to consider here is this statement::

```
BTN S4 /H {C f}
```

Now remember that a "C" is actually a "SHF c" and therefore if you're going to be holding down the LSHF key, then the "f" will actual become an "F". You couldn't after all produce an "f" character from your keyboard if you were keeping a SHIFT button held down. Just something to be aware of.

The beauty of being able to use /H with more than one character generated can be demonstrated with the following statement, which allows you to select a weapon and then hold the firing of that weapon down:

```
BTN S4 /H Select_Rockets DLY(600) Fire Rem Select secondary weapons then fire
```

NOTES

1. *Now, there is an important point to understand here. And that is the difference between a /H modifier, and a /A modifier. Normally when you press a key on your keyboard, let's say for example the "a" key, then an "a" is produced, then there's a short delay, and then a series of "a" characters are produced whilst you keep your finger down on the key. ie.:* a *some_delay* aaaaaaaaaaaaaaaa *This is the same as using a /H modifier. So:*

```
BTN T3 /H a
```

will do exactly this. If I want to get rid of that delay in-between the first and second character, I can achieve this with the /A modifier. So:

```
BTN T2 /A b
```

produces bbbbbbbbbbbbbbbbbb with no delay.

2. *The Rate at which characters are generated with /A statements is determined by the* USE RATE (time_ms) *configuration statement. If such a statement doesn't exist in a file, the compiler will generate a* USE RATE (0) *statement and your default keyboard repeat rate will control this.*

3. *Note that if the /H slash modifier is followed by multiple macros/characters, only the last one is held down, while all preceding macros/characters are produced once. So in this statement:*

```
BTN S2 /H a b c
```

when S2 is pressed and held, only a single "a" and "b" are generated, before the "c" is held down.

4. *You cannot use /H or /A statements after a /R modifier.*

3.5.4 Slash code rules and hierarchy

When you use slash modifiers (eg, /T, /P, /U), you need to understand that there are some basic rules as to what order they are allowed in.

3.5.4.1 Slash code rules

1. They must be placed after the button/switch codes (i.e. BTN S2 /H).
2. They must be placed before the 1st macro (i.e. BTN S2 /H SomeMacro) except /T which can appear within a statement several times.
3. There must be a single space before and after the forward slash codes (see preceding example).
4. They must appear in a specific order when more than one type is used in a statement (*see "Slash Code Hierarchy" below*).
5. The /U /M /D modifiers must be on separate lines to each other.
6. The /I /O modifiers must be on separate lines to each other.

3.5.4.2 Slash code hierarchy

When using multiple slash codes for configuring a button or switch it is important to use the correct hierarchy. The ordering of slash codes would proceed as follows:

1. The /U /M /D modifiers if present would precede all other slash codes.
2. The In/Out (/I and /O) would then be next.
3. The Toggle (/T) modifier precedes /P /R modifiers.
4. The Press and Release (/P and /R) modifiers are always after these.
5. The Hold(/H) and Auto-repeat (/A) modifiers are always last.

Let's look at some examples:

```
BTN S4    /U  /I  macro6
              /O macro7
          /M /P macro8 /R macro9
          /D /T a
             /T /H b c
             /T /A d DLY(30) e DLY(30) f

BTN S2    /I  /T /P macro1 /R macro2
            /T /P macro3 /R macro4
          /O  /H macro5
```

NOTES

There is no problem leaving a statement blank You don't need to use Null characters as you had to with the original TM HOTAS. So this statement:

```
BTN S1   /I Drop_Stores
         /O
```

will not generate a compiler error - it is fine.

3.6 DELAY AND REPEAT STATEMENTS

Statement modifiers

DLY (Delay)
RPT (Number)

where:

Delay is a time in milliseconds (1 second = 1000 milliseconds) and values from 0 to approximately 82800000 are valid. (*For your information, 82800000 milliseconds equates to 23 hours!*)

Number is a value who's maximum changes depending on the length of the button macro, and can range between 2 and 127.

Let's look at some examples to demonstrate how these statements work.

3.6.1 DLY( ) statements

```
BTN T6 1 DLY(60) 1 DLY (60) 2 Rem Request Vector For Recovery TAW
```

would result in "1 1 2" being generated, with a 60 millisecond delay between each character. So why would you want to have delays in between characters? Flight sims and games are becoming so complex that it is common to see a specific group of characters required to perform an action. In a flight sim, this is commonly for communications and often through a menu system. For example in Falcon 4:

```
VectorToHomePlate = q DLY(60) q DLY(60) 6
```

If we had the macro without the delay statements, like this:

```
VectorToHomePlate = q q 6
```

then there's a very good chance that the sim won't pick up these 3 characters, because the controller will produce them too quickly, and the sim is busy with its own calculations and graphics routines. So placing delay statements in between the characters slows their generation down a little, which is more representative of how you would do this with fingers on a keyboard, and thus the sim will see the characters. The situation becomes a little more complex depending on whether a USE RATE (time) configuration statement is present (*see later notes*). A USE RATE (time) statement determines the rate at which continuous keyboard characters are sent from the controllers.

So:

```
USE RATE (60)
BTN S1 q q 6
```

is exactly the same as:

```
USE RATE (0)
BTN S1 q DLY(60) q DLY(60) 6
```

Note that the compiler defaults to a USE RATE(0) statement if one isn't present in the file.

When you are using DLY statements, you don't need to keep your finger pressed on a button for the complete statement to execute. For example:

```
BTN S2 h DLY (2000) e DLY (2000) l DLY (2000) l DLY (2000) o
```

results in "hello" being spelt out, with a 2 second pause between each character. You wouldn't need to keep your finger on button S2 for the whole statement to execute. An interesting point to note here though is that if you press S2 twice within 2 seconds, the results will be:

```
"hheellllloo"
```

This is a feature of this HOTAS Cougar - it's ability to process statements in parallel. To prevent this, either be careful how you use the Cougar, or enclose the statement within < > brackets.

3.6.2 RPT( ) statements

```
BTN S2 RPT(6) c  Rem 6 chaffs please … like right now would be good!
```

Using the RPT command instructs the characters or macros that follow the RPT (nnn) to repeat a specific number of times, where 'nnn' specifies the number of times the characters or macros are to be repeated. The item repeated would be the one which **immediately** follows the RPT command, (i.e., a repeat count could be applied to any single character or a group of characters and commands enclosed in parentheses).

So for example in the above statement, pressing button S2 generates 6 "c" characters at a rate determined by any USE RATE(x) statement in the file.

Some more examples:

```
BTN S1 RPT(10) a b
```

would send 10 "a" characters followed by 1 "b" character.

```
BTN S1 RPT(10) (a b)
```

would send "a b" ten times in a row.

```
BTN S1 /A RPT(10) (a DLY(60)) DLY(2000)
```

would send 10 "a" characters with 60 millisecond delays between each "a" character, and then there'd be a 2 second delay, and then the whole statement would repeat.

RPT counts can be 'nested' - for example:

```
BTN S1 RPT(10) a RPT(10) b
```

would be a valid statement, and:

```
BTN S1 RPT(10) (a RPT(10) b)
```

would be as well.

If you have a macro like this:

```
Macro1 = a b c
```

and a statement like this:

```
BTN S2 RPT (3) Macro1
```

then when S2 is pressed you will get:

a a a b c

To avoid this, either enclose the macro or the characters in its definition with parentheses brackets, i.e.:

BTN S2 RPT (3) (Macro1) *or*

BTN S2 RPT (3) Macro1 *where*

Macro1 = (a b c)

3.7 CHARACTER GROUPING - USING BRACKETS

Statement modifiers

```
BTN T2 (a b c)
BTN T3 {a b c}
BTN T4  /P <a b c>
        /R d
```

Before we look at these brackets for character grouping, there's a very important rule when using them. The brackets **( ) { }** and **< >** are **reserved** TM syntax statements.
You cannot therefore assign them directly to buttons/macros. Instead you must use the *shifted* statements:

(= SHF **9**
) = SHF **0**
{ = SHF **[**
} = SHF **]**
< = SHF **,**
> = SHF **.**

So a macro statement:

Left_ToeBrake = <

will generate a compiler error, and the correct statement is:

Left_ToeBrake = SHF .

3.7.1 ( ) Parentheses

Parentheses are used in various TM syntax statements, eg. DLY ( ), RPT ( ), USB ( ), KU ( ), Digital Type statements etc. to group statements together. We saw how they were used for example in this complex RPT and DLY statement from the previous page:

```
BTN S1 /A RPT(10) (a DLY(60)) DLY(2000)
```

The important thing to note here is that with the original TM syntax, placing brackets around a group of characters or macros forced them to repeat. This is no longer the case with the HOTAS Cougar, as we now have the /A modifier, and so parentheses are just used to group macros/characters together for statements. Note that when parentheses are used in TM statements, the compiler interprets these statements as being the same: DLY (20), DLY(20), DLY (20) etc.

One final point before we leave parentheses. If you want to generate a "(" or ")" character in your joystick file, then do not use these characters directly, ie.

```
Macro1 = (
Macro2 = )
```

The parentheses and other brackets (such as < > { }) are reserved TM characters as they are used in various statements and affect how such statements behave.

So if you have:

```
BTN S1 Macro1
```

then although it will compile/download fine, it won't generate a "(" character. Instead you must use the actual key presses you would need to press to generate these characters. ie.

```
Macro1 = SHF 9
Macro2 = SHF 0
```

See the section on Macros and Macro rules. Remember that you can use Korgy to ensure that you generate the correct TM syntax for the keys/characters you want to generate.

3.7.2 { } Curly brackets

The next type of grouping uses "Curly Brackets". This grouping allows a set of characters to be generated as though they were all being held down at the same time. Curly bracket groups are treated as a single entity for processing. For example:

BTN S4 {a b c}

is sent as "a" press, "b" press, "c" press, "a" release, "b" release, "c" release, as if you had pressed and held one key after the other, then released them in the same order. (Think of what you do when you have to use CTRL ALT DEL ☺)

Also note that a curly bracket group may be used in conjunction with a /H code, in which case all keys in the group will be held until the button is released. For example:

BTN S4 /H {CTL ALT DEL} is perfectly acceptable (*and often desirable!*)

ADVANCED NOTES

1. *You cannot use DLY statements within curly brackets.*

2. *The implementation of { } statements on USB devices differs from the previous gameport TM HOTAS. On the previous TM HOTAS, then characters within a { } statement were produced in the order in which they appeared. So:*

BTN S4 {a b c}

resulted in "a" press, "b" press, "c" press, "a" release, "b" release, "c" release. On USB though, the order of the characters is actually as per their USB definitions, and are thus for characters in alphabetical order. So:

BTN S4 {c b a}

will result in "a" press, "b" press, "c" press, "a" release, "b" release, "c" release. However, in reality they are all produced ***at the same time, within the same frame*** *so the situation is more equivalent to pressing "a" "b" and "c" on a keyboard at the same time, and releasing them at the same time. If you really wanted to separate them out so that they occurred in a specific order, then you'd need to use* KD *and* KU *statements, like this:*

BTN S4 KD(c) KD(b) KD(a) KU(c) KU(b) KU(a)

3_7_3 < > Angle brackets

The angle brackets are new syntax for the HOTAS Cougar. Everything within angle brackets forces the controllers to complete those statements before any others. For example, consider the following statement:

```
BTN H1D q DLY(60) q DLY(60) 6 Rem Vector for Homeplate in Falcon 4
```

It could be disastrous if another button is pressed in the meantime, as it could change this Falcon 4 communication request into a different request entirely. So by adding < > to the statement:

```
BTN H1D <q DLY(60) q DLY(60) 6> Rem Vector for Homeplate
```

when Hat 1 is pressed down, forces the entire statement to be generated before any others can be interpreted. This can be very useful when trying to prevent "sticky" keys:

```
BTN T4 /P < DLY(2000) KD (b) >
          /R KU (b)
```

Pressing button T4 will always ensure that the /P statement is executed fully, even if button T4 is released before the execution is completed. So the "KU (b)" character will only be produced after the "b" key is pressed. If this wasn't the case, then the "KU (b)" action would be generated as soon as T4 was released, and the "b" key would never be released, resulting in it appearing as though it was "stuck" pressed down.

NOTES

1. *You can't use < > within { } brackets, so:*

```
BTN S2 { < a b > }
```
will generate a compiler error.

2. *You cannot cascade < > brackets, so:*

```
BTN T4 < < a b > >
```
will generate a compiler error.

3. *The < > don't have to enclose the whole statement, so:*

```
BTN S1 a b <c d > e f
```
is fine.

4. *If we have a statement like this:*

```
BTN S2 /H <a b c>
```

then as soon as the "a" and "b" characters have been produced, the statement effectively becomes a held down "c" character and the forced part of the

statement is completed. Remember that a /H *statement involving several characters results in the last character being held down.*

5. *The way that < > statements work is that if any other statement is currently being executed, then it will continue to be executed. So if you're holding down a button elsewhere which has a* /H *statement on it, then this will continue to work, whilst you press the button with the < > statement. Any other button presses result in their statements being added to the buffer memory, to be executed when the forced < > statement has completed execution. It therefore follows that one should be careful how you use < > statements, and certainly if you include long* DLY ( ) *statements within a < > statement, there exists the possibility to really stall your controllers. See the troubleshooting section later.*

ADVANCED NOTES

We had, as one of the examples earlier:

BTN S4 KD(c) KD(b) KD(a) KU(c) KU(b) KU(a)

This statement puts each KeyDown *and* KeyUp *into 6 separate frames. If there's a default* RATE *statement then a user may want to have the* KeyUp *statements occurring at the same time, and not at a speed defined by a* RATE *statement. This can be done like this:*

BTN S4 KD(c) KD(b) KD(a) KU({c b a})

so now the KeyUp *statements are put into 1 frame and therefore occur faster.*

3.8 WORKING WITH AND DEFINING DIRECTX (DIRECT INPUT) BUTTONS

If you've ever used a simple joystick, with just a trigger on it, then in a flight sim you'd have found that the trigger would have fired your guns/missiles. It does this, not because you've programmed it to do this, but because Windows told the game "*This joystick has a trigger - you decide what to use it for.*" The trigger is just a button. Such a button is called a DirectX button - a button who's function is assigned by the game.

Configuration statement

USE *button_identifier* or *logical_flag* AS DX*n*

Command syntax

BTN *button_identifier* DX*n*

where:

button identifier is H1U, T6, S2 etc.
logical flag is X1 to X32 (*discussed later in the reference book*)
n is 1 to 28

The HOTAS Cougar consists of 10 analogue axes (with the new rudders), 28 buttons, and a POV HAT (Hat 1). When the Cougar is in Windows mode, the buttons can be assigned controls within a game or flight-sim. This occurs because Windows informs the sim what the capabilities of the controllers are - ie. what axes, buttons, POV, etc are present. When we program a file, we place configuration statements in the joystick file that allow us to determine what to report to the sim.

In programmed mode, by default, none of the buttons will be seen by the sim as DirectX buttons. So we need to inform the sim if we want to assign them. The most common button assigned as a DirectX button is the Trigger:

```
USE TG1 AS DX1
```

We don't need to do anything else - the trigger will be seen by the sim and allocated a function, usually firing guns and/or missiles. Other statement examples are:

```
USE H1U AS DX2
USE X4 AS DX3   Rem assigned to a logical flag - see later notes
USE T4 AS DX5
```

We can also actually program any DXn statement into other button statements, and do all sorts of clever things:

```
BTN S2 /H a DLY(2000) DX2
```

In this example, when I hold down button S2, an "a" character will be produced, followed by a 2 second delay, and then DX2 will be held down. In this case we're "almost" defining button S2 as DX2 because we've got no USE statement defining it elsewhere. I use the term "almost" because this is not the same as having a:

```
USE S2 AS DX2
```

statement, which would result in DX2 being generated as soon as S2 is pressed. In our example DX2 is generated only after "a DLY(2000)" has executed. And only then will S2 generate DX2.

3.8.1 USE ALL_DIRECTX_BUTTONS

We've seen how to assign individual buttons as DirectX buttons. This is a good time to introduce you to the configuration statement, USE ALL_DIRECTX_BUTTONS.

Configuration statement
USE ALL_DIRECTX_BUTTONS

This assigns all buttons as DirectX buttons. Thus a file with this statement in it would assign all of the non-programmed buttons as DirectX buttons, whilst still allowing curves to be modified, the mouse to work etc. etc. Note that any Default Options that you have setup in Foxy will be ignored.

So you could have a very simple file:

```
Rem Set up all the buttons as DirectX buttons
USE ALL_DIRECTX_BUTTONS
Rem Assign the mouse to the microstick - see later notes in the ref book
USE MTYPE A3
Rem And we'll also assign Hat1 on the joystick as a POV hat
USE HAT1 AS POV
```

and this would give you on downloading a joystick and throttle that has Hat 1 as a POV, all buttons as DirectX, and a mouse on the microstick. You could then start to add button statements and build a file gradually.

NOTES

1. *The new syntax replaces the* PORT Bx IS *statements of the original TM HOTAS.*

2. *The default DirectX button assignments for the Cougar when in Windows mode are:*

DX	BTN ID	DX	BTN ID	DX	BTN ID	DX	BTN ID
1	TG1	8	H2R	15	H4U	22	T4
2	S2	9	H2D	16	H4R	23	T5
3	S3	10	H2L	17	H4D	24	T6
4	S4	11	H3U	18	H4L	25	T7
5	S1	12	H3R	19	T1	26	T8
6	TG2	13	H3D	20	T3	27	T9
7	H2U	14	H3L	21	T2	28	T10

with Hat1 *defaulting to a POV (Point Of View) Hat.*

3. *If you have a* USE ALL_DIRECTX_BUTTONS *statement in your joystick file, and you've programmed one of your controller's buttons, then it won't be setup as a DirectX button. The way the Compiler converts a* USE ALL_DIRECTX_BUTTONS *statement is to assign the default DirectX buttons as follows:*

```
BTN TG1 /H DX1
BTN S2 /H DX2
```

etc. Now if in your file it comes across a button statement say for BTN S2, then it won't set that button up as a DirectX button. This is therefore a great way to have the majority of your buttons assignable in a game, but also allowing you to program some of them for your own requirements.

4. *If you do use a* USE ALL_DIRECTX_BUTTONS *statement, and any combination of* USE MTYPE *and/or* USE HATn AS POV (*see later notes*)*, then these statements* ***must*** *come* ***before*** *any button statements, or the Compiler will generate an error. Remember that you should always try and structure your file so that configuration statements appear before any button and axis statements.*

5. *The* USE MTYPE *statement, (which we'll be discussing later in the reference book), can assign left and right mouse buttons to T1 and T6 on the throttle depending on what type of* MTYPE *statement you insert. If you do have a* USE MTYPE *statement with a* USE ALL_DIRECTX_BUTTONS*, then if the* MTYPE *statement assigns any buttons as mouse buttons, they will not be assigned as DirectX buttons.*

The MTYPE (A1 *to* A5) *assigns mouse buttons as follows:*

A1: *T1 = Left mouse button, T6 = Right mouse button*
A2: *T1 = Right mouse button, T6 = Left mouse button*
A3: *T1 = Left mouse button*
A4: *T6 = Left mouse button*
A5: *Doesn't assign any mouse buttons.*

6. *If you select a different HAT as a POV control, (we'll be covering this later in the reference book) for example:*

```
USE HAT3 AS POV
```

then the compiler will not assign DirectX buttons to its positions.

ADVANCED NOTES

1. *If we take the example we used earlier:*

```
USE TG1 AS DX1
```

then what the compiler does with this is to translate the statement into:

```
BTN TG1   /P KD (DX1)
          /R KU (DX1)
```

Knowing that we can use the KD *and* KU *syntax to separate out the* KeyDown *and* KeyUp *parts of the DirectX button, let's look at a funky example:*

```
BTN TG1   /I  /A KD (DX1) DLY(50) KU (DX1) DLY (200)
          /O /H DX1
```

Now if in your sim the Guns were assigned to DirectX button 1, then with the above statement, with S3 out, the guns would fire, and with S3 in, the guns would fire intermittently.

3.9 USING KD, KU AND USB CODES

There are occasions when you want more control over KeyDown and KeyUp events, or want to be able to send a direct code that defines a special key - say only available on a non-US keyboard. This can be achieved using the following:

Command syntax

KD(*Keyboard character/DX buttons/Mouse buttons*)
KU(*Keyboard character/DX buttons/Mouse buttons*)

USB(*Key_eventHID code*)

3.9.1 KD, KU

All of these statements are designed to provide programmable control of what you do when you press a key on your keyboard. And obviously when you press a key, you press the **K**ey **D**own (**KD**) and then take your finger off it and allow the key to come up (**KU**). Sometimes you will want to be able to program other characters to be generated in between these 2 events. And you can do so like this:

```
BTN H1U KD(UARROW) DLY(20) KU(UARROW)
```

In this example, pressing HAT1 up results in a cursor up arrow key being pressed for 20 ms, and then released. KD and KU can be used on any key - you just need to use the correct TM syntax.

It is also possible to combine keys within KD, KU statements.

```
BTN T4 KD(a b c) DLY(20) KU(a b c)
```

You can also use KD and KU on DirectX buttons, Mouse buttons and logical flags (*logical flags are covered later in the reference book*). For example with the mouse left button (MOUSE_LB):

```
BTN T6 KD(MOUSE_LB) DLY(2000) KU (MOUSE_LB)
```

Pressing button T6 emulates the mouse left button being pressed for 2 seconds and then released.

3.9.2 USB programming

It is also possible to be able to send the actual USB code to support any syntax that's not currently supported in the TM default syntax. This can be useful for non-US keyboard layouts. USB codes are given in Appendix 3 at the end of this reference book and each key's code is suffixed with either a "D" to represent the KeyDown event, or a "U" to represent the KeyUp event. For example:

```
BTN T3 /P USB (D51) /R USB (U51) Rem 'Down arrow'
BTN T4 USB (DE1 D04 UE1 U04) Rem 'Shift a'
```

The USB codes in these examples are producing keyboard key presses, and they do so putting them into the memory structure into different frames. If you want to generate these key presses so that they happen at the same time, then enclose them as normal within { } brackets. For example:

```
BTN T4 USB (DE1 D04) DLY (2000) USB ({UE1 U04}) Rem 'Shift a'
```

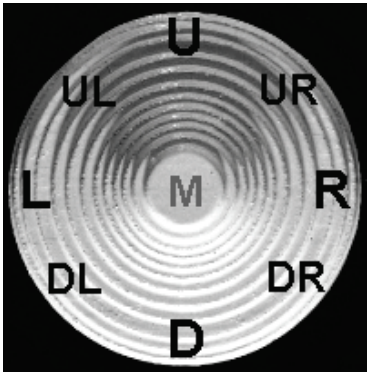
results in the Left Shift and 'a' keys being released at the same time. I should point out though that the difference in timing between frames is very small – approximately 30 milliseconds, so you're not likely to see the difference.

4. HAT Programming

4.1 PROGRAMMING THE JOYSTICK HATS

4.1.1 Programmable positions on a hat

Hats have 9 programmable positions, although in general, you'll only program the main 4 primary directions. For HAT 1 for example these would be:



BTN H1U Look_up
BTN H1R Look_right
BTN H1D Look_down
BTN H1L Look_left

But the corner positions are also programmable:

BTN H1UL View_UL
BTN H1UR View_UR
BTN H1DL View_DL
BTN H1DR View_DR

and the middle position:

BTN H1M View_forward

It is important to note that all these programmable positions are separate and that the corner positions are not by default a product of whatever's programmed on the positions either side. So if I program the main up, down, left and right positions to be the keypad 8, 2, 4 and 6, then pushing the hat into the UR position is not going to result in an 8 and 6 being held down, or a 9 being produced. It won't do anything. Of course this relies on your ability to move the hat into the corner position directly.

4.1.2 4-way vs. 8 way hats: USE HatID FORCED_CORNERS

If I wanted to force the corner positions to generate a combination of the positions either side of them, then I could use the configuration statement:

Configuration statement
USE *HatID* FORCED_CORNERS

where: *HatID* is either HAT1, HAT2, HAT3, HAT4
e.g. USE HAT1 FORCED_CORNERS

(*Oh by the way, there is a statement that you can use to make it easier to use the corner positions more reliably: see USE HatID_SENSITIVITY(nnnn) later in this reference book*)

A HAT can be assigned as one for programming as normal, or as the mouse, the POV (Point Of View control), the arrow keys, or as the Key Pad numbers. There are 4 special configuration statements that can be used to set up a hat for different purposes. And they are:

Configuration statement
USE *HatID* AS MOUSE (rate) [- *optional modifiers*] USE *HatID* AS POV [- *optional modifiers*] USE *HatID* AS ARROWKEYS [- *optional modifiers*] USE *HatID* AS KEYPAD [- *optional modifiers*]

where:

HatID is either HAT1, HAT2, HAT3, HAT4, RADIOSWITCH

(*The RADIOSWITCH although it doesn't look like a HAT, actually is, consisting of buttons T2 (Up), T3 (Down), T5(Left), T4 (Right). It differs slightly from the normal 4 hats in that it effectively has a built in FORCED_CORNERS statement on it.*)

Rate is 1 to 127 and applies only to the USE *HatID* AS MOUSE (rate) statement.

[-*optional modifiers*] Can be used with these statements to modify the behaviour of the hat. They consist of:

REVERSE_UD, REVERSE_LR, FORCED_CORNERS, NOHOLD, KP5.

Note that not all of these optional modifiers can be used with all hat statements - see the relevant sections below, or just use Foxy's Composer.

4.1.3 Controlling the mouse with a HAT.

Configuration statement

USE *HatID* AS MOUSE (rate) [- *optional modifiers*]

Rate: - the speed of the mouse is 1 to 127
Optional modifiers permitted: REVERSE_UD, REVERSE_LR

e.g. USE HAT1 AS MOUSE (2)

Quite simply, this assigns control of the mouse onto HAT 1. The value in brackets, in this case 2, determines the rate at which the mouse will travel across the screen – a low value results in a sluggish mouse, and a high value in a fast one. If the hat is moved into a corner position, then the mouse will move diagonally.

If we wanted to reverse the Up and Down directions of the mouse we can do this like this:

USE HAT1 AS MOUSE (2) - REVERSE_UD

and similarly to reverse the Left and Right directions we can do this like this:

USE HAT1 AS MOUSE (2) - REVERSE_LR

You can use the two REVERSE_*types* together if you want. So:

USE HAT1 AS MOUSE (2) - REVERSE_UD, REVERSE_LR

is a perfectly valid statement.

NOTES

It's also possible to have the mouse controlled from a hat, at the same time as from the microstick, or anywhere else for that matter. It is also possible to have the hat control the mouse for a given position of the dogfight switch (/U, /M, /D) *on the throttle, and/or S3* (/I, /O) *position. This is covered in more detail in the Mouse programming section: Understanding the Mouse Device and the Microstick.*

4.1.4 Setting up a HAT as a Point Of View (POV) HAT

Configuration statement

USE *HatID* AS POV [- *optional modifiers*]

Optional modifiers permitted: REVERSE_UD, REVERSE_LR

e.g. USE HAT3 AS POV

When you use the controllers in Windows mode, or if you have no HAT 1 statements (BTN H1x) in your joystick file, HAT 1 defaults to operating as a standard POV HAT. A POV HAT is a special 8-way HAT that in many sims, can be assigned control by the sim. For example, in Falcon 4, if you don't program HAT 1, it will behave as a POV HAT for view control. In a similar manner to previous statements, you can also reverse the directions of the POV in the up/down and left/right directions with:

USE HAT4 AS POV - REVERSE_UD
USE HAT1 AS POV - REVERSE_LR
USE HAT3 AS POV - REVERSE_UD, REVERSE_LR

You can also program POV positions directly using POVU, POVD etc, even if you haven't assigned a hat as a POV. You can program any position that you want (*and omit what you don't want*). It's like DX buttons - they're there, but need programming to make them active. So long as the hardware detects a POV being present on the type of handle you're using, you can program the POV positions. See the Notes section below.

NOTES

We've seen how easy it is to assign a hat as a POV control. But it's also worth pointing out that the POV Hat positions can be programmed directly in a file. The syntax for the positions of the POV Hat are similar to the positions on a normal Hat. They are:

POVU, POVD, POVL, POVR, POVUL, POVDL, POVUR, POVDR

and you would program with them like this:

BTN T5 POVL
BTN T4 POVR

4.1.5 Using a HAT to emulate the keyboard arrow keys

Configuration statement

USE *HatID* AS ARROWKEYS [- *optional modifiers*]

Optional modifiers permitted: REVERSE_UD, REVERSE_LR, NOHOLD

e.g. USE HAT2 AS ARROWKEYS

It is very common in flight sims to need to be able to program the arrow keys onto a hat, and that's exactly what this statement does. The arrow keys will be held down for the duration of the hat press. In a similar manner to previous statements, you can also reverse the directions of the arrow keys in the up/down and left/right directions with:

USE HAT3 AS ARROWKEYS - REVERSE_UD
USE HAT4 AS ARROWKEYS - REVERSE_LR
USE HAT1 AS ARROWKEYS - REVERSE_UD, REVERSE_LR

Note that if you move the hat into a corner position, then the hat will generate the arrow keys either side of that corner position - it has its own built in FORCED_CORNERS modifier.
If you don't want the arrow keys to be held down, use the NOHOLD modifier with this statement, like this:

USE HAT3 AS ARROWKEYS - NOHOLD

This will produce single arrow keys when the hat is moved into its different positions. As with the other optional modifiers, it can be used in conjunction with them, like this:

USE HAT1 AS ARROWKEYS - REVERSE_UD, REVERSE_LR, NOHOLD

4.1.6 Using a HAT to emulate the numerical keypad keys

Configuration statement

USE *HatID* AS KEYPAD [- *optional modifiers*]

Optional modifiers permitted:
REVERSE_UD, REVERSE_LR, FORCED_CORNERS, NOHOLD, KP5

eg. USE HAT4 AS KEYPAD

It is also common to want to emulate the numerical keypad with a HAT. With the above statement then, HAT 4 will generate the numerical keypad numbers, with the corner positions of the hat (UL UR, DL, DR) generating the "7 9 1 3" keys respectively.

The problem with the numerical keypad is that it's behaviour varies considerable between different flight sims and games. Not only that, different sims behave differently depending on whether the Num LOCK is engaged, or they may force it to be either on or off. However, in most cases, it usually works if one treats the numerical keypad as just that - generating numbers.

In a similar manner to previous statements, you can also reverse the directions of the arrow keys in the up/down and left/right directions with:

```
USE HAT1 AS KEYPAD - REVERSE_UD
USE HAT2 AS KEYPAD - REVERSE_LR
USE HAT3 AS KEYPAD - REVERSE_UD, REVERSE_LR
```

We've said that the statement generates the numerical keypad keys. Of course, there's one key missing, and that's the "5" key (KP5). Some sims assign the KP5 to centre something useful, and you can instruct the compiler to generate a KP5 at the hat's centre position, with the - KP5 syntax as follows:

```
USE HAT4 AS KEYPAD - KP5
```

We can also force the corner positions,

```
USE HAT4 AS KEYPAD - FORCED_CORNERS
```

which instructs HAT 4 to behave so that if it is pushed into say the UR corner position, instead of generating a "KP9", to generate instead "KP8" and "KP6" together, ie. the Up and Right assigned characters.
This may not make much sense when thinking of the hat generating numbers as such, but are interpreted in some sims and games correctly.

If you don't want the keypad keys to be held down, use the NOHOLD modifier with this statement, like this:

```
USE HAT3 AS KEYPAD - NOHOLD
```

This will produce single keypad keys when the hat is moved into its different positions.

You can even combine all of these optional modifiers if you so wish:

```
USE HAT4 AS KEYPAD - REVERSE_UD, REVERSE_LR, FORCED_CORNERS,
NOHOLD, KP5
```

4.1.7 How the Compiler converts USE HatID AS statements

This section is for advanced users only, with an unhealthy appetite for detail!

You will need to have read other sections of the reference book to understand these following advanced notes, but for those who love to get technical, I'm going to discuss a little how the Compiler actually converts the various USE HAT*n* AS *something_useful* statements. We'll start of with the USE HATx AS MOUSE statement.

USE HAT1 AS MOUSE (2) is converted by the compiler into:

```
USE HAT1 FORCED_CORNERS
BTN H1U /P MSY(2-) /R MSY(2+)
BTN H1R /P MSX (2+) /R MSX (2-)
BTN H1D /P MSY (2+) /R MSY (2-)
BTN H1L /P MSX (2-) /R MSX (2+)
```

Similarly the Compiler converts: USE HAT1 AS MOUSE (2) - REVERSE_UD

to:
```
USE HAT1 FORCED_CORNERS
BTN H1U /P MSY(2+) /R MSY(2-)
BTN H1R /P MSX (2+) /R MSX (2-)
BTN H1D /P MSY (2-) /R MSY (2+)
BTN H1L /P MSX (2-) /R MSX (2+)
```

and finally: USE HAT1 AS MOUSE (2) - REVERSE_UD, REVERSE_LR

to:
```
USE HAT1 FORCED_CORNERS
BTN H1U /P MSY(2+) /R MSY(2-)
BTN H1R /P MSX (2-) /R MSX (2+)
BTN H1D /P MSY (2-) /R MSY (2+)
BTN H1L /P MSX (2+) /R MSX (2-)
```

Now we'll look at the ARROWKEYS conversion: USE HAT2 AS ARROWKEYS

is converted to:

```
USE HAT2 FORCED_CORNERS
BTN H2U /H UARROW
BTN H2R /H RARROW
BTN H2D /H DARROW
BTN H2L /H LARROW
```

Similarly: USE HAT2 AS ARROWKEYS - REVERSE_UD, NOHOLD

is converted to:

```
USE HAT2 FORCED_CORNERS
BTN H2U DARROW
BTN H2R RARROW
BTN H2D UARROW
BTN H2L LARROW
```

Notice how the Up and Down positions are swapped, and that the inclusion of the NOHOLD modifier removes the /H from the statements.

And finally we'll look at the assignment as a KEYPAD: USE HAT4 AS KEYPAD

is converted to:

```
BTN H4U /H KP8
BTN H4R /H KP6
BTN H4D /H KP2
BTN H4L /H KP4
BTN H4UR /H KP9
BTN H4DR /H KP3
BTN H4DL /H KP1
BTN H4UL /H KP7
```

and similarly: USE HAT4 AS KEYPAD - REVERSE_UD, NOHOLD

is converted to:

```
BTN H4U KP2
BTN H4R KP6
BTN H4D KP8
BTN H4L KP4
BTN H4UR KP3
BTN H4DR KP9
BTN H4DL KP7
BTN H4UL KP1
```

Notice how the corner positions are swapped as well when we reverse the UD direction, and that the inclusion of the NOHOLD modifier removes the /H from the statements.

The same is true of REVERSE_LR:

```
USE HAT4 AS KEYPAD - REVERSE_LR
```

which is converted to:

```
BTN H4U /H KP8
BTN H4R /H KP4
BTN H4D /H KP2
BTN H4L /H KP6
BTN H4UR /H KP7
BTN H4DR /H KP1
BTN H4DL /H KP3
BTN H4UL /H KP9
```

Again notice how the corner positions are swapped as well. When we force the corner positions, with:

```
USE HAT4 AS KEYPAD - FORCED_CORNERS
```

the compiler converts the statement to:

```
USE HAT4 FORCED_CORNERS
BTN H4U /H KP8
BTN H4R /H KP6
BTN H4D /H KP2
BTN H4L /H KP4
```

Finally:

```
USE HAT4 AS KEYPAD - KP5
```

result in the compiler generating an extra:

```
BTN H4M KP5
```

statement. Note now that there's no /H slash modifier. The "KP5" is a non repeating character. That's quite enough of all of that. Let's move on …. if you haven't already done so!

5. Configuration statements

5.1 INTRODUCTION

In the beginning of this reference book, we discussed the layout of a joystick file. Now although statements can appear anywhere in a joystick file, we set out an area before the main programming statements, for configuration statements. You've already been introduced to some configuration statements, such as the USE MDEF configuration statement. It's just that we didn't confuse the issue earlier on by taking time out to explain what we mean by configuration statements. So let's do that here, now that you should be a little more comfortable with programming your controllers.

Configuration statements apply to the whole file - they're not statements programmed to specific controller positions, although some can be programmed directly onto button statements. They tell the compiler how to setup the controllers for your sim. They include statements that tell the compiler which macro file to use for its macro definitions, what rate you'd like characters generated at, what axes you may want to disable etc. etc.

Many of the configuration statements are described in detail elsewhere in this reference book, alongside the areas that they are relevant to. I will therefore concentrate here on those that haven't been discussed in detail elsewhere.

The syntax has changed though for the HOTAS Cougar when it comes to configuration statements - so here's the Golden Rule:

All configuration statements begin with either USE or DISABLE

All logical programming configuration statements begin with DEF

This is different to the previous Thrustmaster syntax. Oh - and don't get hung up at this stage about the term "Logical programming" that has reared its ugly head in several places. It's mentioned for sake of completeness and it is pertinent to the areas where I've brought it up, but it's a topic we'll cover right at the end of the reference book, as it really is for the hardcore programmers amongst you.

5.2 MDEF - MACRO DEFINITION FILE

Configuration statement

USE MDEF *macro_filename*

This statement is only needed if a joystick file contains macros (*which is to be encouraged*). The *macro_filename* in such a statement is the name of the macro file with or without its extension (.tmm).

For example, if I have a joystick file: Janes WW2 Fighters.tmj and its macro file is Janes WW2 Fighters.tmm, then the MDEF statement would be:

USE MDEF Janes WW2 fighters

It is important that both files are in the same directory, and by default this is Foxy's Files folder. Although it is possible to design our software so that they don't need to be, I think it's a practical convention to stick with, as per the original TM HOTAS.

NOTES

1. *Long file names with spaces are allowed for macro file names.*

2. *It is irrelevant whether you put the extension onto the end of the filename in the* MDEF *statement. So both of these are fine:*

USE MDEF Janes WW2 fighters
USE MDEF Janes WW2 fighters.tmm

3. *Macro filenames, joystick filenames and macro names are not case sensitive*

Mig Alley.tmm

is the same macro file as:

mig alley.tmm

4. *In the original TM HOTAS, you could actually use multiple* MDEF *statements, and hence use macro definitions from different macro files. This is not the case for the HOTAS Cougar.*

5.3 RATE

Apart from the remaining configuration statements below, the majority of configuration statements relate to the programming of the axes, and are covered in that section of this reference book.

Configuration statement

USE RATE (*nnnn*)

Command syntax

RATE (*nnnn*)

where *nnnn* is a time in milliseconds (1000ms = 1 second). Determines the rate at which characters repeat. If omitted, the compiler will default to a USE RATE (0) statement resulting in characters being produced at the default keyboard typematic rate. The **larger** the rate value, the **slower** the rate at which characters are generated. Rate values between 0 and 655350 (*just over 10 minutes!)* are permitted.

It is also possible to change the RATE value in real time, by programming it onto a button:

```
BTN S4 /I RATE (100)
       /O RATE (0)
```

or in an axis statement:

```
ANT 2 5 RATE (0) RATE (30) RATE (60) RATE (90) RATE (120)
```

(*see later notes on programming digital axis statements*)

ADVANCED NOTES

The HOTAS Cougar produces characters in groups called frames. Frames are generated at somewhere around 30ms intervals with a RATE(0) *(or without a RATE) statement. Within a frame, the Cougar can generate 16 characters. If more characters are produced and that frame consists already of its 16 character maximum, further characters are shunted to the next frame. The* RATE *value effectively determines the time interval between frames.*

5.4 S3_LOCK AND S3_ UNLOCK

Configuration statement

USE S3_LOCK

Command syntax

S3_LOCK
S3_UNLOCK

Normally BTN S3 on the joystick is used so that when it is pressed in, all /I statements are generated, and when it is released, all /O statements are generated.

A USE S3_LOCK statement means that if S3 is pressed, then the Cougar will only use the /I statements. When it is pressed again, it'll shift over to the /O statements.

If you wanted to be able to toggle between the two states then you could use something like this:

BTN S2 /T S3_LOCK /T S3_UNLOCK

You don't need to have a USE S3_LOCK configuration statement present in a file to use the direct S3_LOCK, S3_UNLOCK statements on a button. So what's the difference between USE S3_LOCK and just S3_LOCK? USE S3_LOCK applies to the whole file and is active as soon as the file is downloaded and enabled. Whereas S3_LOCK on a button statement only applies once that button has been pressed.

NOTES

1. *If you assign a different button/hat as S3 (see the next section) then these statements will apply to that button, but you use the same syntax.*

2. *You cannot use hold slash modifier* (/H) *with* S3_LOCK *statements. So:*

BTN S1 /H S3_LOCK
BTN S4 /H *Some_macro* S3_LOCK

will both generate compiler errors.

5.5 ASSIGNING A DIFFERENT BUTTON FOR /I, /O WITH SHIFTBTN

Configuration statement

USE *button_identifier* AS SHIFTBTN

Command syntax

SHIFTBTN (*button_identifier)*

Examples:

```
USE S4 AS SHIFTBTN
BTN T6 SHIFTBTN (T10)
```

Determines which button to use instead of the S3 button for selecting /I statements. If this statement is missing, and it will probably be rarely used, then button S3 is used for /I, /O statements.

5.6 USE HAT SENSITIVITY - HAT CORNER SENSITIVITY

It can sometimes be very difficult finding the hat corner positions easily. (eg. H4UL) Remember that the hats are in essence just 4 way switches. Therefore, because the controllers process everything so quickly, you may often see one of the macros generated from either side of the corner position first before hitting the corner position. You can reduce the sensitivity of the hat to help eliminate this problem with:

Configuration statement

USE *HatID*_SENSITIVITY (*nnnn*)

where:

HatID is one of the 4 hats: HAT1, HAT2, HAT3, HAT4
nnnn is a value from 0 (most sensitive) to 1000 (least sensitive). The *nnnn* is actually a delay in milliseconds. For example:

```
USE HAT1_SENSITIVITY (100)
```

would mean that all statements on HAT1 would only occur after pressing the hat position for 100ms, giving more time to close the positions either side of a corner position.

NOTES

You cannot use /T *slash modifiers on any of that Hat's programmable positions if you have a* USE *HatID*_SENSITIVITY (*nnnn*) *statement setup for that Hat.*

So: USE HAT1_SENSITIVITY (*60*)
BTN H1U /T a /T b /T c *would generate a compiler error.*

5.7 USE T1 SENSITIVITY

If you find that you're constantly accidentally pressing the T1 button on the microstick, then you can make it less sensitive.

Configuration statement

USE T1_SENSITIVITY (*nnnn*)

where:

nnnn is a value from 0 (most sensitive) to 1000 (least sensitive). The *nnnn* is actually a delay in milliseconds, and represents the delay before the T1 button press is recognised. This feature is implemented for people who find that they're accidentally pressing T1 too easily.

Here's an example:

USE T1_SENSITIVITY (1000)

would mean that T1 would only be seen as being pressed after a 1 second delay.

NOTES

You cannot use /T *slash modifiers with a* BTN T1 *statement if you have a* USE T1_SENSITIVITY (*nnnn*) *statement in your file.*

5.8 USE FOXY GRAPHIC AND README

Configuration statement

USE FOXY GRAPHIC *imagefile*
USE FOXY README *textfile*

These two configuration statements are ignored by the compiler and are only used by Foxy for its purposes. When Foxy opens up your files, it scans through the files, and if it comes across a USE FOXY GRAPHIC *imagefile* statement, it will load up the *imagefile* into the Image Viewer. This is useful when you're distributing your files, and you have a graphical layout showing what macros are assigned to what buttons and hats. An example of this statement is:

USE FOXY GRAPHIC Total Air War.bmp

Graphic files permitted are bitmaps (.bmp), jpegs (.jpg) or gifs (.gif).

Similarly, the USE FOXY README configuration statement, instructs Foxy to load up a text file into the Template Editor, which can be useful to others, or as a reminder to yourself, describing how you've designed your files, what settings need to be made in the flight sim for them to work etc. etc. An example of this statement is:

USE FOXY README Total Air War.rtf

Text files permitted are either simple text files, which have the .txt extension, or Rich Text Files, which have the extension .rtf. Rich text files can have coloured text, formatting, different fonts etc, and so can be much easier to read.

NOTES

The graphic and text files must be located in the same directory as your joystick and macro files. By default this is in Foxy's Files folder.

5.9 NULLCHR - NULL CHARACTER

Configuration statement

USE NULLCHR *character*

A null character is a character in a statement that doesn't generate any output. The default null character is the **^** caret. So why bother I hear you ask? Well

some statements require a fixed number or parameters for the statement to be valid. Some good examples are Digital Type statements which we're going to come onto in the next section. For example, this statement:

```
RDDR 3 L ^ R
```

programs the rudders to generate a held down "L" character when the left rudder is pushed forward, and a held down "R" character when the right rudder is pushed forward. When the rudder is at rest and centred, I don't want it to produce anything, so I've added the default null character, "^" for the centre position. I couldn't just leave it blank like this:

```
RDDR 3 L  R
```

because this statement requires 3 characters/macros after the "RDDR 3" otherwise the compiler would generate an error. So think of null characters then as *place holders* where you need to have a character, but you don't want to produce anything.

Coming back then to this configuration statement, the default null character as I've said is the caret (**^**). If you want to use a different character, then you can do so with statements like these:

```
USE NULLCHR TAB
USE NULLCHR z
```

If a caret is used in a game, you can still assign it to the controls indirectly by entering SHF 6 into the assignment, ie. BTN S1 SHF 6

NOTES

1. *With the original TM controllers, you were always advised to never leave a statement line blank if it was in your joystick file, and to add a null character there, like this:*

```
BTN S2 /U Fire_Missile
       /M ^
       /D  ^
```

This is not the case with the Cougar. There's no problem with having:

```
BTN S2 /U Fire_Missile
       /M
       /D
```

in your joystick file.

2. *The null character actually produces the USB* (00) *code. This generates nothing, and so it's also possible if you prefer to set up a macro in your macro file like this:*

```
Do_Nothing = USB(00)
```

and then use that in your statements:

```
RDDR 3 L Do_Nothing R
```

Of course, it's so much quicker, easier and neater to use the caret ^ and hence the reason for its existence.

3. *You cannot use chorded keys with the* USE NULLCHR *statement. So these will generate compiler errors:*

```
USE NULLCHR SHF F1
USE NULLCHR ALT  p
```

5.10 KEYBOARD (AZERTY, QWERTY)

Configuration statement

USE KEYBOARD *Keyboard type*

Where *Keyboard type* is either: AZERTY or QWERTY.

If you're using a French AZERTY keyboard, and the game you're programming your files for remaps keys to match your keyboard layout, so that they're not performing the functions correctly in your game, then add a USE KEYBOARD AZERTY statement to your file to see if that fixes it. See the Key Tester section for more information.

NOTES

Don't bother using a USE KEYBOARD QWERTY *statement - the compiler always defaults to this when compiling files. It's unnecessary to have this in a file.*

5.11 USING PROFILES FROM THE COUGAR CONTROL PANEL - USE PROFILE

Configuration statement

USE PROFILE *Profile* (Calibration Mode)

where:

Profile: is a profile created by the Cougar Control Panel and saved with the extension .tmc in the Cougar software's Profiles folder.
Calibration mode: is either AUTO or CUSTOM. Use this to specify whether you want the Cougar Control Panel to use the profile with autocalibration data or a custom calibration that you've set up.

Examples:

USE PROFILE Crimson Skies.tmc (AUTO)
USE PROFILE Mechwarrior 4 (CUSTOM)

As we shall soon see with the axis programming, it is possible to change the axes so that they are swapped around, disabled, have different response curves etc. etc. This can get quite complicated to set up with various statements. However, if you've used the Cougar Control Panel to set up profiles, and saved them, then it can be much easier just to use them. Using a saved profile also has the added advantage that deadzones can be incorporated into the axes response curves, which cannot be adjusted programmatically. Furthermore from my experience, downloads are faster if using a profile compared to using DISABLE or USE AXES_CONFIG statements (*see later notes.*)

5.11.1 Some more discussion on profiles

I'm going to spend a little time here discussing tmc files, ie. profiles, and why they're inherently a very good idea to include in all your joystick files. First off then, profiles are created using the Cougar Control Panel (CCP). They contain all the information you can set up in the CCP relating to the axes, such as any axis mapping, dead zones, curves etc.
Now if you download a file that affects any of this axis data, or use a file that allows you to say change your joystick curves whilst flying, then when you exit the sim, all that information stays stored in the controllers. Remember, this is a driverless system and all information is stored in the controllers. If you then go and download a different file for a different sim, that doesn't reset this axis information, then you will inherit the values from the previous sim, because they're already stored in the Cougar. You have 2 options therefore to get round this if it's something that causes a problem for you. You can either use a USE PROFILE statement in each of your joystick files, setting it say to the

DEFAULT.tmc file (*the default profile created by the CCP the first time it is run*) or a profile of your choice for that sim, or in Foxy, you can from the Download menu, ask the Compiler when it downloads a file to first of all reset the axes by applying your chosen profile, each time the file is downloaded. I hope that makes sense.
Finally, because a profile contains calibration data, this is the reason why you have to inform the compiler whether you want to use the calibration data in the profile with your sim, or whether you'd like to use the profile with Autocalibration enabled.

NOTES

1. *Foxy and the compiler will initially assume that the profiles are in the Cougar software's Profiles folder. This is where all profiles created with the Cougar Control Panel are saved. By default this is:* C:\Program Files\Hotas\Profiles*. If when a file is being compiled/downloaded containing a* USE PROFILE *statement, then the compiler:*

   (a) Will look for the profile in its default Profiles directory.
   (b) If the profile exists, it will use this one (and not try to look anywhere else).
   (c) If the profile does not exist, it will look in the same directory as the joystick file, i.e. Foxy's Files folder.
   (d) If the profile still does not exist, it will generate an error.

2. *Profiles have the file extension* **tmc***. It doesn't matter whether you have that extension with the* USE PROFILE *statement. So:*

   USE PROFILE Crimson Skies.tmc *is equivalent to*
   USE PROFILE Crimson Skies

3. *If you're using Foxy to open up a zip of someone else's files, and that zip contains a profile, then all the files in the zip are extracted to Foxy's Files folder. You can choose whether you want to move the profile to the Cougar software's Profiles folder although Foxy won't do this for you - you'll need to use Explorer. I recommend keeping all profiles in the HOTAS Profiles folder* (*usually* C:\Program Files\HOTAS\Profiles).

4. *A profile contains principally the following information about the Cougar axes: Mapping data, Axis directions, Centre positions, Calibration data, Dead Zone information, Curves data, Trim information, Disabled axes data, and the Apply Axis Disable/Enable View option.*

5.12 CONFIGURATION STATEMENTS DESCRIBED ELSEWHERE IN THE REFERENCE BOOK

Some of the configurations statements are beyond the scope of this chapter as they need explaining in respect to other statements. The following configuration statements are discussed elsewhere:

Configuration statement	Described in
USE *Btn* AS DXn	Section 3.8: Working with and defining DirectX (Direct Input) buttons
USE ALL_DIRECTX_BUTTONS	Section 3.8.1: Assigning all the buttons as DirectX buttons
USE HAT AS MOUSE, POV, ARROWKEYS, KEYPAD	Section 4.1: Programming the Joystick HATS
USE CURVE	Section 6.3: Response curves and (CURVE)
DISABLE AXIS	Section 6.5: Disabling Axes
USE SWAP	Section 6.6: Axis Mapping (SWAP)
USE REVERSE	Section 6.7: Reversing the direction of an axis
USE AXES_CONFIG	Section 6.8: The USE AXES_CONFIG statement
USE MTYPE	Section 7.2: USE MTYPE - the simplest way of assigning the mouse to the microstick
USE Axis_Identifier AS Mouse_Axis	Section 7.3.1: Assigning other axes to mouse axes
USE ZERO_MOUSE	Section 7.5: Prevents stuck mouse movement with custom mouse statements and /I, /O
DISABLE MOUSE	Section 7.7: Disabling the default assignment of the mouse to the microstick
USE SCREEN_RESOLUTION	Section 7.8.1: Defining the screen resolution
DEF Xn	Section 8.2: Defining logical flags and their button statements

6. Axis Programming

6.1 BASIC PRINCIPLES

6.1.1 Understanding the difference between Analogue and Digital

We're now going to take a look at how you can program the various Cougar axes, both digitally, or modify their analogue behaviour. Before we do this though, let's explain the difference between an analogue and a digital axis, as many people find these terms confusing.

Most joysticks on the market these days work in exactly the same way mechanically. Inside, they contain two potentiometers, or **pots** as they're referred to. If you have a radio/Hi-Fi that has a knob on it that you turn to control the volume, what you're actually turning is a pot that varies its resistance as you turn it.

In a joystick, these pots are connected to the joystick perpendicular to each other, to measure the left/right motion of the joystick, the x axis, and the forwards/backwards motion, the y axis. So a joystick has two principal axes along which it can travel, the x and y axis. The position of the joystick as you move it can be determined in terms of these 2 axes, ie. how far along the x axis the joystick has moved, and how far along the y axis. The pots therefore give a *range* of values as you move your joystick, and as such, they are termed *analogue* devices, as opposed to *digital* devices, which are either on or off, like the keys on your keyboard, or buttons on your joystick.

Now, if you're still with me and haven't moved onto the next section, then let's complicate the issue here a little. In a perfect world, pots would give you very precise and stable values from them, at all their positions of rotation. But pots can suffer degradation due to wear and dust. *I know I do!*. The effect of these is to sometimes produce a slightly fluctuating value, or worse, the odd "jump" (or "spike") in value. With the HOTAS Cougar, the signals from the pots aren't fed directly through to your flight-sim. A digital processor inside the Cougar reads in the values from a pot, and then filters out erroneous values, to give a more precise and stable value. So although the pots are *analogue*, the *signals* from them are processed *digitally*.

We'll soon be looking at what we can do with the axes on the HOTAS Cougar, and one thing that we can do is to program them digitally. With the above in mind, it's worth explaining this further, because it's not immediately apparent how we can program an analogue axis digitally. Let's say for arguments sake, that the throttle produces analogue values in the range 0 to 100. It is possible to divide the axis up into say 5 bands, so band 1 is equivalent to readings from 0 to 19, band 2

is equivalent to readings from 20 to 39 etc. And we can devise a statement that says "When we're in band 1, produce an 'a' character, in band 2 a 'b' character" etc. We refer to this as programming an axis with a digital type statement, which I will explain further in the following sections.

6.1.2 The Cougar Axes

The HOTAS Cougar (Joystick, Throttle, Rudders with toebrakes) has 10 physical analogue axes. These axes can be treated as:

- purely analogue devices, *(assuming that your game and DirectX supports them*) and can therefore assign controls to them
- purely digital devices, so that they can be programmed to generate keyboard characters only
- or a combination of the above two.

Furthermore, we can affect the analogue axes by:

- removing them completely
- applying different response curves
- applying trim values
- reversing the direction of the axes
- mapping them to other axes, both as a default for the whole file as well as based on the position of the dogfight and S3 switches

One of the major strengths of the Cougar is just how much can be done with these axes. However, this can get very complicated, very quickly! So to try and simplify our understanding of what we can achieve in a Joystick file in terms of handling and programming these axes, we need to first define the **6 Digital Type statements**. These are used to program the axes digitally to generate keyboard characters. After that we'll be in a good position to look at what we can do with the analogue axes as the sim sees them.
Before we go any further, let's define the syntax for these 10 physical axes:

TM Syntax	Axis
JOYX	Joystick X
JOYY	Joystick Y
THR	Throttle
RNG	Range
ANT	Antenna
MIX	Microstick X*
MIY	Microstick Y
LBRK	Left Toe Brake
RBRK	Right Toe Brake
RDDR	Rudder

* Note: Microstick X, Y are different to Mouse X, Y

NOTES

1. *With the original TM HOTAS an axis was either analogue or digital - ie. it was seen as a default axis by a game and assigned a function (eg. TQS Throttle = Thrust in game) or programmed digitally to generate keyboard characters. This is not the case with the new HOTAS Cougar. By default the axes are seen as analogue but if you program them digitally, then they will be* ***both*** *analogue* ***and*** *digital. If a pure digital axis is required the axis in question should be disabled.*

2. *You do not need to change anything in Control Panel, Gaming Options Applet if you want to use an axis only digitally. This was the case with the previous TM controllers, but isn't the case with the new Cougar.*

3. *With all digital axis statements:*

- *You can use /U, /M, /D, /I, /O statements*
- *Any analogue axis modifications (curve, map, reverse) etc do not affect digital statements. They stay on the physical axis, and linear.*

6.2 DIGITAL TYPE STATEMENTS

In this section we're going to look at how to program the axes digitally so that they produce keyboard characters. Programming any of the axes to produce keyboard characters is achieved by using one of the 6 available Digital Type Statements. The easiest way to understand these 6 types is to look at an example of each type, and see what characters are produced.

Note: *Not wanting to complicate things straight away, but just keep in the back of your mind that it's not just characters that must be used within these Type statements. Logical flags, mouse statements and axis curve statements for example can also be used.*

6.2.1 Type 1: repeating character generation

A type 1 statement has the following syntax:

Axis identifier	Digital Statement Type	Number of characters or macros (max. 50)	Up character or macro	Down character or macro	Centre character or macro *(optional)*	FORCE_ MACROS *(optional)*
eg. ANT	1	10	u	d	c	- FORCE_MACROS

or as it would appear in a joystick file:

ANT 1 10 u d c - FORCE_MACROS

Rotating the ANT knob Clockwise and then Anticlockwise would result in the following characters being generated:

u u u u u c u u u u u d d d d d c d d d d d

We're not restricted to using single characters in digital statements, so we could have macros in the macro file defined as:

```
Chaff_Flare =  c DLY(30) f
Getting_desperate = RPT(20) (c f)
```

and have a Type 1 statement for the RNG knob in the joystick file:

RNG 1 5 Chaff_Flare Getting_desperate

Note that the Number of characters now defines the total number of characters generated (*excluding any centre character*) for the **full** travel of the axis. This is different to the original TM Type 1 statement, where the Number of characters defined the number of characters generated from one end of the axis travel to the centre character, and then from the centre character to the other end of travel. The reason for the change in syntax is because now we're making the centre character optional in the statement. For example:

RNG 1 6 u d

produces the following characters on rotating the RNG knob:

u u u u u u d d d d d d

Note that excluding a centre character **is different** to using a null character (*default is* ^) for the centre character. So the statement:

RNG 1 6 u d ^

produces:

u u u *dead-zone* u u u d d d *dead-zone* d d d

The Null character "^" results in nothing being produced … a type of dead-zone if you will. Before we leave this point, the Number of characters **must be an even number** *if* a centre character is provided, otherwise if there's no centre character, it can be anything. The reason for this I hope should be obvious: if the number of characters required is 20 and there's a centre character, then you want to be able to have 10 either side of the centre character.

6.2.1.1 Understanding the - FORCE_MACROS modifier

This modifier is optional, and can be used only with Type 1, 2, 5 and 6 digital statements. I'm going to use a Type 1 statement to explain the significance of it, but the following applies equally to the other type statements it can be used with.

Let's say we have the following:

RNG 1 50 u d

Rotating the RNG knob from one end of its travel to the other, will produce 50 "u" characters, or 50 d" characters, depending on the direction you move it in. Now if you move the RNG knob quickly, and test the result in Notepad or Foxy's Key Tester, you won't see 50 characters produced. You might see 10 to 20 but not 50. So what's going on? Is this a bug - I mean the Cougar is meant to be able to process statements very quickly, isn't it? Yes, the Cougar does process statements very quickly, and in parallel, and that's actually the reason why you're seeing this effect. What happens when you rotate the RNG knob quickly, isn't that any of the characters are being missed as such. It's that their key presses and key releases are being processed in parallel and if up to 16 characters are being processed, the computer will likely see those as just one character press in one frame. We can change this behaviour though by forcing the computer to see each character, like this:

RNG 1 50 (< u >) (< d >)

and that's essentially what the - FORCE_MACROS modifier does. It surrounds each character/macro in a digital Type statement with (**<** *character/macro* **>**). Just be careful how you use this modifier though - if you're forcing statements from one position then other statements elsewhere don't get produced until the forced statement is processed. Also, make sure that any macros you use in a digital statement with a - FORCE_MACROS modifier don't have the force modifiers (the angle < > brackets) in their definition. You cannot cascade force modifiers and you will do so if you have:

```
Macro_1 = < a b c>
Macro_2 = d
```

RNG 1 50 Macro_1 Macro_2 - FORCE_MACROS

as the compiler would convert this to:

RNG 1 50 (< < a b c> >) (< d >)

NOTES

1. *You cannot have:* RNG 1 3 a b c d e f

but you can have RNG 1 3 (a b c) (d e f)
or RNG 1 3 ABC_macro DEF_macro

and in your macro file: ABC_macro = a b c
DEF_macro = d e f

2. *You can use* /U, /M, /D, /I, /O *with all digital statements, for example:*

```
RNG  /U 1 3 F1 F2
     /M 1 5 (SHF UARROW) (SHF DARROW)
     /D  /I 1 6 e t F5
         /O 1 4 [ ] KP5
```

3. *You can also use* /P, /R *and* /H *within your macros or within brackets directly in your digital type 1, 2, 5 and 6 statements. For example:*

```
RNG 1 3 Macro_1 Macro_2
```

and in your macro file:

```
Macro_1 = /P a /R b
Macro_2 = /H d  Rem But why you'd want to do this I don't know!
```

4. *You cannot use* /T *slash modifiers within Type 1 (or* ***any*** *digital axes) statements.*

5. *A Type 6 statement is a special case of a Type 1 statement. So these statements produce identical responses from the Antenna knob* (u u c u u d d c d d):

```
ANT 1 4 u d c
ANT 6 5 (0 20 40 60 80 100) u d c
```

6.2.1.2 Important considerations when using FORCE_MACROS

ADVANCED NOTES

It would seem that it's a good idea to use the FORCE_MACROS *all the time from the above explanation, but this isn't true by any means. The first issue you need to think about is the effect on not only the response of other programmed hats and buttons on your controllers, but also the response from the axis you've added* FORCE_MACROS *to. Let's look at an example, which also helps me explain something else.*

Consider this Type 2 Digital Axis statement (yes, I know I haven't covered it - it's in the next section but this is an easy one to understand.)

ANT 2 26 a b c d e f g h i j k l m n o p q r s t u v w x y z

Now it doesn't take a genius to understand that ignoring the syntax, basically what this statement does is to program the ANT knob on the throttle to generate the alphabet.

(Apologies if your alphabet is different to mine! And to my American audience, we pronounce 'z' as 'zed' on this side of the pond, not 'zee' but that's completely immaterial. I've no idea why I brought that up here. I'm rambling again.)
Now if you add this to your joystick file, and download it, then when you rotate the ANT knob, the characters in the alphabet are produced, and you can use Foxy's Key Tester to view them. What you'll notice is that the characters are produced very quickly indeed. If you then modify the statement by adding the FORCE_MACROS *modifier, like this:*

ANT 2 26 a b c d e f g h i j k l m n o p q r s t u v w x y z - FORCE_MACROS

and do the same, you'll notice that the output from the ANT knob is very much slower - in fact it's at least 4 times slower. This is because for each character, say the "a" character, the compiler converts it into: < KD(a) KU(a) >. *Each of these 4 components goes into their own "frame".*

I'd better explain frames before I continue. When the Cougar wants to output characters or programmed statements, it sends them out every 30ms. If you like this is the Cougar's frame rate. Each frame can contain several characters - it's not one character per frame (remember the Cougar processes in parallel, up to 32 macros at the same time). So in the first example I gave for the ANT and the alphabet, one of the reasons it generates the alphabet characters so quickly is that it sends several of the characters in one frame depending on how fast you rotate the ANT knob. You actually see this in Foxy's Key Tester. Many of the Key Down *events go out together, followed in the next frame by their* Key Up *events. I'll come back to this…*

Anyway, coming back to the FORCE_MACROS *modifier then, now the "a" character gets converted into* < KD(a) KU(a) >. *These 4 components of the statement* (<, KD(a), KU(a) and >) *go into separate frames, and hence the whole alphabet is going to be generated* ***much*** *slower.*

Whilst we're with this ANT *statement, I'm going to head off topic for a little bit and explain another feature of the Cougar - this ability to send multiple characters in the same frame, and one of the effects of this relating to the order in which characters are produced. Let's look at the statement again:*

ANT 2 26 a b c d e f g h i j k l m n o p q r s t u v w x y z

You would think that with this statement that rotating the ANT *knob one way is going to produce the alphabet keys alphabetically, and rotating in the opposite direction, the exact opposite. Well try it out and see what happens when you rotate the knob quickly. You will see in Foxy's Key Tester that actually some of the characters appear out of order. This is most noticeable rotating the knob anticlockwise, and for me this generates:*

"x y z u v w p q r s t m n o j k l f g h i a b c d e"

You'll also notice that in the Up/Down/Keycode events window, that the characters are being produced as a group of Down *characters, followed by a group of* Up *characters, i.e.* Down (z y x) Up (z y x) *instead of* Down (z) Up (z) Down (y) Up (y) *etc. So why are we seeing this odd order when rotating the* ANT *knob in this direction? Is this a bug? Nope, it's not. It's a feature of how USB devices send "keyboard keys." With the older TM controllers, characters were sent using the PS2 standard which is as individual key down and key up events. With USB devices, the way it works is like this: characters aren't sent out as such, rather the operating system scans a keyboard buffer and notes which keys are pressed and which aren't. So when the Cougar modifies the buffer to indicate that the z, x and y keys are pressed, the operating system picks this up in its scan and then outputs the characters in a predefined order, which for characters is alphabetical order. So in the key test, although you've instructed the* ANT *knob to send the characters z, x and y in that order, if the* ANT *knob is rotated fast enough so that the Cougar places the* z, x *and* y *into the same frame, then the computer sees these 3 characters as being pressed at the same time and displays them in their alphabetical order, i.e.* x, y *and then* z.

Of course when we add the FORCE_MACROS *modifier, then we always get the alphabet in the correct order for the rotation direction of the* ANT *knob, because each character will never be placed in the same frame as another character being generated from the* ANT *knob. But we generate them much slower.*

I hope you understood most of that. So think carefully as to when you need to use the FORCE_MACROS *modifier for your flight sim and when you don't. Remember the golden rule - it's not how it behaves in Windows that matters - you must go and try it out in your sim and use that as your final testing ground. One final note: If you test out that statement* (*without the* FORCE_MACROS *modifier*) *then you may notice that some characters don't get generated when the* ANT *knob is rotated quickly - indeed you may get some odd Up characters like* SHF *and* CTL *generated. This is something that we are aware of and doesn't seem to be a bug in the Cougar. We think it relates to the Microsoft keyboard driver somehow getting overloaded to the point where it reinitialises itself, resulting in these erroneous* Up *characters. It's impossible for the Cougar to generate these erroneous key up characters with that statement.*

6.2.2 Type 2: custom character sequence, fixed regions

A type 2 statement has the following syntax:

Axis identifier	Digital Statement Type	Number of characters or macros (max.50)	Sequence of characters and/or macros and/or logical flags	FORCE_ MACROS *(optional)*
eg. ANT	2	5	a b c d e	- FORCE_MACROS

or as it would appear in a joystick file:

ANT 2 5 a b c d e - FORCE_MACROS

Rotating the ANT knob Clockwise and then Anticlockwise would result in the following characters being generated:

b c d e d c b a

Just as with the Type 1 statement, notice that each character is only produced once (*but logical **flags** stay on – see later notes*). Again this will also be true if macros are used instead of single characters. Also, note that with the ANT knob starting from its fully counterclockwise position, turning it clockwise will generate a "b" as the first character, and not as you may have thought an "a". This is a different behaviour compared to a Type 1 statement. It means that if you keep rotating the ANT knob through its full range of positions several times, you effectively cycle through the output characters as follows:

b c d e d c b a b c d e d c b a b *etc. etc.*

so that you're not producing:

a b c d **e e** d c b **a a** b c d **e e** d c b a *etc. etc.*

Note that unlike the original TM syntax, the Number of characters or macros can be odd or even. Obviously if you're assigning a statement to something like the ANT knob you'll probably prefer to use an odd number so that the centre position corresponds with the centre detent.

You can also use macros with a type 2 digital statement in a similar manner:

RNG 2 5 Emcon-1 Emcon-2 Emcon-3 Emcon-4 Emcon-5

and in the macro file:

```
Emcon-1 = e DLY(40) 1
Emcon-2 = e DLY(40) 2
Emcon-3 = e DLY(40) 3
Emcon-4 = e DLY(40) 4
Emcon-5 = e DLY(40) 5
```

Now I know that we haven't discussed yet what logical flags are, but if you do get into that side of TM programming, then just be aware that it is also possible to directly use logical flags in a Type 2 statement, including logical flag toggles:

RNG 2 4 X1 X2* X3* X4

I'll discuss this is more detail in the logical coding section. As well as this, single characters, macros and logical flags can be mixed in a type 2 statement as follows:

ANT 2 5 a Emcon-2 c ^ X1

6.2.2.1 Understanding the - FORCE_MACROS modifier

See the section 6.2.1.1 for the Type 1 statement for an explanation of this modifier.

NOTES

1. *You can use* /U, /M, /D, /I, /O *slash modifiers with all digital statements. However, be careful if you mix these modifiers with any assigned in logical statements (see later notes), as you may get some strange behaviour from your controllers. For example (this is for advanced users only!):*

```
ANT /U 1 6 a b c
    /M 1 6 d e f
    /D 2 3 (DLY(5000) X1) X2 X3
BTN X1 /U a
       /M b
       /D c
```

On the /D *position,* X1 *could generate an "a", "b" or "c" depending on whether the dogfight switch changed position during the 5 second delay.*

2. *You cannot use* /U, /M, /D, /I, /O ***within*** *the statement (this applies to all Digital Type statements). So this would generate a compiler error:*

RNG 2 3 a b macro_1

where macro_1 = /I KP1 /O KP2

3. *Let's say that you've programmed:*

ANT /I 2 3 SM1 SM2 SM3
/O 2 3 SM4 SM5 SM6

where: SM1 = a
SM2 = /H b
SM3 = c
SM4 = d
SM5 = /H e
SM6 = f

Now let's say that you have the S3 switch in the out position, and the antenna knob in the middle section (generating a held "e"), and now you presses the S3 switch. This will result in the "e" being broken so that it is no longer held. The "b" will automatically then be held down, and when S3 is released, the held "e" is remade.

4. *You can use* /P, /R, /H *with Type 1, 2, 5 and 6 statements.*

6.2.3 Type 3: held character generation

A type 3 statement has the following syntax:

Axis identifier	Digital Statement Type	Left character	Centre character	Right character
Eg. RDDR	3	l	c	r

or as it would appear in a joystick file:

RDDR 3 l c r

Pushing the left rudder pedal would result in a held down "l" character being produced, in exactly the same way that held down characters are produced with the /H modifier.

NOTES

1. *The axis isn't divided up into 3 equal regions but instead more like below, otherwise the centre region is likely to have too much travel:*

Left region	Centre region	Right region
/H l	c	/H r

2. *You can use logical flags with a Type 3 statement if you wish.*

3. *If you don't want to use a centre character, use a null character* (^):

RDDR 3 l ^ r

6.2.4 Type 4: pulsed character generation

A type 4 statement has the following syntax:

Axis identifier	Digital Statement Type	Pulse rate (ms)	Left character or macro	Centre character or macro	Right character or macro
eg. RNG	4	1000	l	c	r

or as it would appear in a Joystick file:

RNG 4 1000 l c r

A pulsed character is a character that's produced once every x milliseconds, a bit like a lighthouse beam or strobe light. Type 4 statements are new to TM controllers. With the Range statement above, when the Range knob is turned to the left (*or clockwise looking at it face on*) then an "l" character is produced every 1000 milliseconds (ie. once every second). Conversely, rotating it in the opposite direction, a single "c" character is produced at the centre position, and then an "r" character every second.

NOTES

1. *Macros and logical flags can also be used instead of single characters.*

2. *If you don't want to use a centre character, use a null character:* ^

RNG 4 60 l ^ r

3. *The Pulse rate is a value in milliseconds between 0 and 82800000 (which is 23 hours!)*

6.2.5 Type 5: custom character sequence, variable regions

A type 5 statement has the following syntax:

Axis identifier	Digital Statement Type	Number of regions (max. 50)	Region widths (as percentages)	Sequence of characters and/or macros and/or logical flags	FORCE_ MACROS *(optional)*
eg. THR	5	4	(0 20 45 70 100)	a b c d	- FORCE_MACROS

or as it would appear in a joystick file:

THR 5 4 (0 20 45 70 100) a b c d - FORCE_MACROS

Type 5 character statements look a little more complex than the others at first, but in essence they are a special case of a Type 2 statement. Remember that a Type 2 statement has its character sequence distributed evenly across the axis travel. A type 5 statement divides up the axis into regions or bands, and then assigns its characters in sequence into those bands.

In the above example, there are 4 bands setup:

- 0 to 20% of axis travel: produces an "a" character
- 21 to 45%: "b" character
- 46% to 70%: "c" character
- 71% to 100%: "d" character

In every other aspect a Type 5 character statement follows the same rules and restrictions as a Type 2 statement. Now, a digital statement can exist with an analogue axis. So we could have a default analogue throttle, but using a Type 5 statement, get characters generated at any point in the travel of that axis. So it would be very easy to introduce reverse thrusters or activate your wheel brakes during landing at minimum throttle, on the down position of a throttle statement:

```
THR /U
    /M
    /D 5 1 (0 5) Wheelbrakes
```

And in the macro file I have:

```
Wheelbrakes = /P b /R b
```

So when the dogfight switch is in the down position, and the throttle is in the minimum position, the Wheelbrakes macro will engage the wheel brakes. Note how I don't want any digital statements in the dogfight switch's middle (/M) and up (/U) positions. So the Wheelbrakes macro is only executed when the dogfight switch is in the down position.

6.2.5.1 Understanding the - FORCE_MACROS modifier

See the section 6.2.1.1 for the Type 1 statement for an explanation of this modifier.

NOTES

1. *With the original TM HOTAS, you could program the minimum throttle position with a* BTN MT *statement, but only when the throttle wasn't analogue. The* BTN MT *statement is no longer supported, because of the increased power you get when programming the throttle digitally as well as being able to use it as an analogue throttle. If you want to emulate a* BTN MT *statement, then use the example given above, i.e:* THR 5 1 (0 5) Your_macro_here

2. *As with all digital statements, any curves applied to an analogue axis do not affect the digital statements. They maintain their own linear "curves."*

3. *You can use* /P, /R, /H *with Type 1, 2, 5 and 6 statements but put them in the macro or if you're using them directly in the statement, enclose them with parentheses. So:*

THR 5 1 (0 5) Wheelbrakes

where in your macro file you have the Wheelbrakes macro defined as:

Wheelbrakes = /P b /R b

is fine. And you could also have had:

THR 5 1 (0 5) (/P b /R b) *but:*

THR 5 1 (0 5) /P b /R b *would generate a compiler error.*

6.2.6 Type 6: repeating character generation, variable regions

A type 6 statement has the following syntax:

Axis identifier	Digital Statement Type	No. of regions (max. 50)	Region widths (%)	Up	Down	Centre *(optional)*	FORCE_ MACROS *(optional)*
eg. ANT	6	5	(8 20 40 45 70 80)	u	d	c	- FORCE_MACROS

or as it would appear in a joystick file:

ANT 6 5 (8 20 40 45 70 80) u d c - FORCE_MACROS

A Type 6 digital type statement is essentially the same as a Type 1 digital statement, just that the characters aren't banded into equal bands as they would be with a Type 1 statement. Rather they are placed into bands of your choice.

Unlike Type 1 statements though, **if you include a centre character then the number of regions must be an odd number.**

In the above example, there are 5 bands setup:

- 8 to 20% of axis travel
- 21 to 40%
- 41 to 45%
- 46% to 70%
- 71% to 80%

But just as with a Type 1 statement, moving the ANT knob produces:

u u c u u d d c d d

If the statement was:

ANT 6 5 (8 20 40 45 70 80) u d

then you'd produce:

u u u u u d d d d d

6.2.6.1 Understanding the - FORCE_MACROS modifier

See the section 6.2.1.1 for the Type 1 statement for an explanation of this modifier.

6.2.7 Axis directions: analogue values and digital statements

6.2.7.1 Analogue Axes values

In this section, I'm going to show what analogue values are produced by the axes and when, and give examples for each Digital Type statement, to demonstrate what direction they work in.

Axis position	Analogue value
JOYX - left	0
JOYX - right	max
JOYY - back	max
JOYY - forward	0
THR - back	max
THR - forward	0
RNG - CCW [note 1]	max
RNG - CW	0
ANT - CCW [note 1]	max
ANT - CW	0
MIX - left [note 2]	-
MIX - right	-
MIY- down	-
MIY - up	-
RDDR - left forward	0
RDDR - right forward	max
LBRK, RBRK - up	max
LBRK, RBRK - pressed	0

NOTES

1. *The Range knob is inherently confusing with the direction the digital statements work in. With the* RNG *and* ANT *knobs, the rule is this: You look at the rotary* ***face on****, to determine its counter clockwise* (CCW) *and clockwise* (CW) *directions.*

2. *The Microstick* (MIX, MIY) *isn't reported to Windows as being present as an analogue controller. That doesn't mean that it can't be used as an analogue controller though, as other axes can be mapped onto it.*

6.2.7.2 Type 1 Digital axes statements

JOYX 1 6 r l	As Joystick X is moved from Left to Right, we get "r" characters, and going from Right to Left, we get "l" characters.
JOYY 1 6 f b	As Joystick Y is moved from Back to Forwards, we get "f" characters, and going from Forwards to Back, we get "b" characters.
THR 1 6 f b	As the Throttle is moved from Back to Forwards, we get "f" characters, and going from Forwards to Back, we get "b" characters.
RNG 1 6 r l	As RNG is moved from CCW to CW, we get "r" characters, and going from CW to CCW, we get "l" characters.
ANT 1 6 r l	As ANT is moved from CCW to CW, we get "r" characters, and going from CW to CCW, we get "l" characters.
MIX 1 6 r l	As Microstick X is moved from Left to Right, we get "r" characters, and going from Right to Left, we get "l" characters.
MIY 1 6 u d	As Microstick Y is moved from its Down position Up, we get "u" characters, and going from Up to Down, we get "d" characters.
RDDR 1 6 l r	Left pedal forwards gives "l" characters, left pedal back and right pedal forwards gives "r" characters.
LBRK 1 6 d u	Toe brake pressed Down gives "d" characters, and "u" characters as it's released. (Same applies for RBRK).

6.2.7.3 Type 2 Digital axes statements

JOYX 2 5 a b c d e	As Joystick X is moved from Left to Right, we get "a b c d e" characters, and going from Right to Left, we get "e d c b a" characters.
JOYY 2 5 a b c d e	As Joystick Y is moved from Back to Forwards, we get "a b c d e" characters, and going from Forwards to Back, we get "e d c b a" characters.
THR 2 5 a b c d e	As the Throttle is moved from Back to Forwards, we get "a b c d e" characters, and going from Forwards to Back, we get "e d c b a" characters.
RNG 2 5 a b c d e	As RNG is moved from CCW to CW, we get "a b c d e" characters, and going from CW to CCW, we get "e d c b a" characters.

ANT 2 5 a b c d e — As ANT is moved from CCW to CW, we get "a b c d e" characters, and going from CW to CCW, we get "e d c b a" characters.

MIX 2 5 a b c d e — As Microstick X is moved from Left to Right, we get "a b c d e" characters, and going from Right to Left, we get "e d c b a" characters.

MIY 2 5 1 2 3 4 5 — As Microstick Y is moved from its Down position Up, we get "1 2 3 4 5" characters, and going from Up to Down, we get "5 4 3 2 1" characters

RDDR 2 5 a b c d e — With the Left pedal forwards, and then allowing it to go back as the Right pedal is pushed all the way forwards gives "a b c d e" characters. Moving the Left pedal forwards as the Right pedal moves backwards gives "e d c b a" characters.

LBRK 2 5 a b c d e — Toe brake pressed Down gives "a b c d e" characters, and "e d c b a" characters as it's released. (Same applies for RBRK).

6.2.7.4 Type 3 Digital axes statements

JOYX 3 l ^ r — When Joystick X is Left, we get a held "l" character, and when Joystick X is Right, we get a held "r" character.

JOYY 3 b ^ f — When Joystick Y is Back, we get a held "b" character, and when Joystick Y is Forwards, we get a held "f" character.

THR 3 b ^ f — When the Throttle is Back, we get a held "b" character, and when the Throttle is Forwards, we get a held "f" character.

RNG 3 l ^ r — When RNG is CW, we get a held "r" character, and when RNG is CCW, we get a held "l" character.

ANT 3 l ^ r — When ANT is CW, we get a held "r" character, and when ANT is CCW, we get a held "l" character.

MIX 3 l ^ r — When Microstick X is Left, we get a held "l" character, and when Microstick X is Right, we get a held "r" character.

MIY 3 d ^ u — When Microstick Y is Down, we get a held "d" character, and when Microstick Y is Up, we get a held "u" character.

RDDR 3 l ^ r — With the Left pedal forwards, we get a held "l" character, and with the Right pedal forwards, we get a held "r" character.

LBRK 3 u ^ d — Toe brake pressed Down gives a held "d" character, and when fully up, a held "u" character. (Same applies for RBRK).

6.2.7.5 Type 4 Digital axes statements

JOYX 4 300 l ^ r	When Joystick X is Left, we get a pulsed "l" character, and when Joystick X is Right, we get a pulsed "r" character.
JOYY 4 300 b ^ f	When Joystick Y is Back, we get a pulsed "b" character, and when Joystick Y is Forwards, we get a pulsed "f" character.
THR 4 300 b ^ f	When the Throttle is Back, we get a pulsed "b" character, and when the Throttle is Forwards, we get a pulsed "f" character.
RNG 4 300 l ^ r	When RNG is CW, we get a pulsed "r" character, and when RNG is CCW, we get a pulsed "l" character.
ANT 4 300 l ^ r	When ANT is CW, we get a pulsed "r" character, and when ANT is CCW, we get a pulsed "l" character.
MIX 4 300 l ^ r	When Microstick X is Left, we get a pulsed "l" character, and when Microstick X is Right, we get a pulsed "r" character.
MIY 4 300 d ^ u	When Microstick Y is Down, we get a pulsed "d" character, and when Microstick Y is Up, we get a pulsed "u" character.
RDDR 4 300 l ^ r	With the Left pedal forwards, we get a pulsed "l" character, and with the Right pedal forwards, we get a pulsed "r" character.
LBRK 4 300 u ^ d	Toe brake pressed Down gives a pulsed "d" character, and when fully up, a pulsed "u" character. (Same applies for RBRK).

6.2.7.6 Type 5 Digital axes statements

JOYX 5 5 (0 20 40 60 80 100) a b c d e	As Joystick X is moved from Left to Right, we get "a b c d e" characters, and going from Right to Left, we get "e d c b a" characters.
JOYY 5 5 (0 20 40 60 80 100) a b c d e	As Joystick Y is moved from Back to Forwards, we get "a b c d e" characters, and going from Forwards to Back, we get "e d c b a" characters.
THR 5 5 (0 20 40 60 80 100) a b c d e	As the Throttle is moved from Back to Forwards, we get "a b c d e" characters, and going from Forwards to Back, we get "e d c b a" characters.

RNG 5 5 (0 20 40 60 80 100) a b c d e	As RNG is moved from CCW to CW, we get "a b c d e" characters, and going from CW to CCW, we get "e d c b a" characters.
ANT 5 5 (0 20 40 60 80 100) a b c d e	As ANT is moved from CCW to CW, we get "a b c d e" characters, and going from CW to CCW, we get "e d c b a" characters.
MIX 5 5 (0 20 40 60 80 100) a b c d e	As Microstick X is moved from Left to Right, we get "a b c d e" characters, and going from Right to Left, we get "e d c b a" characters.
MIY 5 5 (0 20 40 60 80 100) 1 2 3 4 5	As Microstick Y is moved from its Down position Up, we get "1 2 3 4 5" characters, and going from Up to Down, we get "5 4 3 2 1" characters.
RDDR 5 5 (0 20 40 60 80 100) a b c d e	With the Left pedal forwards, and then allowing it to go back as the Right pedal is pushed all the way forwards gives "a b c d e" characters. Moving the Left pedal forwards as the Right pedal moves backwards gives "e d c b a" characters.
LBRK 5 5 (0 20 40 60 80 100) a b c d e	Toe brake pressed Down gives "a b c d e" characters, and "e d c b a" characters as it's released. (Same applies for RBRK).

6.2.7.7 Type 6 Digital axes statements

JOYX 6 5 (0 20 40 60 80 100) r l	As Joystick X is moved from Left to Right, we get "r" characters, and going from Right to Left, we get "l" characters.
JOYY 6 5 (0 20 40 60 80 100) f b	As Joystick Y is moved from Back to Forwards, we get "f" characters, and going from Forwards to Back, we get "b" characters.
THR 6 5 (0 20 40 60 80 100) f b	As the Throttle is moved from Back to Forwards, we get "f" characters, and going from Forwards to Back, we get "b" characters.
RNG 6 5 (0 20 40 60 80 100) r l	As RNG is moved from CCW to CW, we get "r" characters, and going from CW to CCW, we get "l" characters.

ANT 6 5 (0 20 40 60 80 100) r l	As ANT is moved from CCW to CW, we get "r" characters, and going from CW to CCW, we get "l" characters.
MIX 6 5 (0 20 40 60 80 100) r l	As Microstick X is moved from Left to Right, we get "r" characters, and going from Right to Left, we get "l" characters.
MIY 6 5 (0 20 40 60 80 100) u d	As Microstick Y is moved from its Down position Up, we get "u" characters, and going from Up to Down, we get "d" characters.
RDDR 6 5 (0 20 40 60 80 100) l r	Left pedal forwards gives "l" characters, left pedal back and right pedal forwards gives "r" characters.
LBRK 6 5 (0 20 40 60 80 100) d u	Toe brake pressed Down gives "d" characters, and "u" characters as it's released. (Same applies for RBRK).

That then covers how to program the various axes digitally. What we're going to move onto now then is to look at the analogue side of the axes, and how we can affect these through various statements. Probably one of the first things you'd like to know then is how to change the response curve of an axis, through programming. Let's take a look ….

6.3 RESPONSE CURVES (CURVE)

All of the 10 axes have by default, linear response curves. That is, if I move the throttle forwards, the values it sends to the simulator are directly related to how far forward I've moved it. ie. for every 10% throttle movement, it's output increases by 10%. It is possible to change the way any of the 10 axes behave, by changing their respective response curves. The response curves for each axis are defined by their sensitivity.
There are 2 statements that can be used to define and change the axes response curves. I'll define these and their terms first, and then explain how to use them:

Configuration statement

USE CURVE (Axis_Identifier, Sensitivity)

Command syntax

CURVE [Slash modifiers] (Axis_Identifier, Sensitivity)

where:

Axis_Identifier is one of the following:

JOYX, JOYY	(together termed **JOYSTICK**)
THR	
RNG, ANT	(together termed **ROTARIES**)
MIX, MIY	(together termed **MICROSTICK**)
LBRK, RBRK	(together termed **TOEBRAKES**)
RDDR	

Sensitivity

is a value that varies from –32 to 32 (*although values beyond 20 result in curves you'll never want to use!*)
Negative numbers (-10 for example) represent reduced sensitivity – great for landing, refuelling, formation flying. Zero (0- duh!) effectively resets the curve to the default linear response (overrides any USE CURVE statement) and positive numbers (10) give increased sensitivity, great for dogfighting in WW2 aircraft for example.

Slash modifiers *(optional)*

/U, /M, /D (Dogfight switch) and /I, /O (Button S3) are permitted

Confused? So am I! So, let's look at some examples, and state what the differences are between the USE CURVE and CURVE statements are.
USE CURVE is a **configuration statement** – (**all** USE statements are). That means, it sits on its own line near the top of a joystick file, and can't be programmed onto a controller position. It is used to define **default axes response curves**. Normally each axis is linear, so setting the Joystick X axis response curve like this:

USE CURVE (JOYX, 0)

would be pretty pointless, as the compiler will do this anyway, assuming that you want linear response curves on the joystick X axis. However if we had the following in the joystick file:

USE CURVE (JOYSTICK, 2)

then this would result in both the joystick X and Y curves being modified to be more responsive. Note how I've used the term "JOYSTICK" to define both the JOYX and JOYY axes, so the compiler will actually send both of the following to the controllers:

USE CURVE (JOYX, 2)

```
USE CURVE (JOYY, 2)
```

So, the USE CURVE statement can be used to modify default axes response curves. The CURVE statement follows the same syntax, but as it isn't a configuration statement, it can be used in button statements, digital axes statements, or as its own special statement, see the examples below. Let's say that we wanted to adjust the sensitivity of the Joystick Y axis based on the position of the Dogfight switch. We can do so with this CURVE statement:

```
CURVE  /U (JOYY, 2)   Rem More responsive for dogfighting
       /M (JOYY, 0)   Rem  Normal
       /D (JOYY, -2)  Rem  Less responsive for landing
```

Similarly, with the microstick:

```
CURVE  /I (MICROSTICK, 2)   Rem More responsive
       /O (MICROSTICK, 0) Rem  Normal
```

And you can mix them:

```
CURVE  /U /I (MICROSTICK, 2) (THR, 2)
          /O (MICROSTICK, 0)
       /M /I (RDDR, -2)
          /O (RDDR, 0) (TOEBRAKES, 2)
       /D (JOYY, -2)
```

The CURVE statement can also be used directly on a programmable position:

```
BTN T7 /P CURVE(JOYX, 3)  Rem Responsive
       /R CURVE(JOYX, 0)  Rem  Normal
```

The CURVE statement can also be used within Digital Axis statements, so if I wanted to I could change the responsiveness of the joystick depending on the position of the throttle. For example:

```
THR 2 5 CURVE(JOYX, -3) CURVE (JOYX, -1) CURVE (JOYX, 0)
CURVE (JOYX, 2) CURVE (JOYX, 5)
```

at low throttle values, the joystick X axis is less responsive than normal, but becomes increasingly more responsive with increased throttle. As a quick aside here, this is actually a lovely example of how the throttle can still behave as an analogue throttle, whilst being programmed digitally. Well I think it's lovely anyway.

NOTES

1. *If an axis is mapped to another axis (see later notes) then its response curve goes with it.*

2. *You cannot setup deadzones with a curve statement - you need to use the Cougar Control Panel for this. If you need these deadzones to be specific to a particular sim, then use the CCP to save these deadzones in a profile and use the USE PROFILE statement in your joystick file. See point 5.*

3. *You cannot have more than one* CURVE *statement in a joystick file. So the second* CURVE *statement here would give a compiler error.*

```
CURVE  /I (MICROSTICK, 2)   Rem More responsive
       /O (MICROSTICK, 0) Rem  Normal

CURVE  /I (ROTARIES, 2)   Rem More responsive
       /O (ROTARIES, 0) Rem  Normal
```

Not to be confused with using CURVE *programmed onto axes, buttons, which you are allowed to use more than once.*

4. *If you're not programming a* CURVE *statement onto a button or axis, then you cannot use* CURVE *on its own without slash modifiers. Instead use a configuration statement. So:*

```
CURVE (JOYSTICK, 10)          will generate an error, whereas:

USE  CURVE (JOYSTICK, 10)     is fine.
```

5. *You can also use a saved profile - see the* USE PROFILE *statement discussed earlier if you want to apply multiple curves to axes for the whole file. This has the advantage of being able to incorporate deadzones.*

6.4 AXIS TRIMMING (TRIM)

Trimming an axis is a means of being able to take your hands off your controllers, and yet the sim sees them as though you're holding them in one position. Let me explain further. Let's say you're cruising along at 15,000ft and for some reason your plane wants to climb when the joystick is centred, so you have to correct this by constantly having to push the joystick forwards. In this example, the TRIM function can be used to allow you to leave the joystick centred, but seen by the sim as though you're pushing the joystick forwards. This isn't limited to the joystick – it applies to all 10 analogue axes.

Command syntax

```
        TRIM (Axis_Identifier, Trim_amount)
and:
        HOLDTRIM (Axis_Identifier)
```

where:

Axis_Identifier is one of the following:

JOYX, JOYY	(together termed **JOYSTICK**)
THR	
RNG, ANT	(together termed **ROTARIES**)
MIX, MIY	(together termed **MICROSTICK**)
LBRK, RBRK	(together termed **TOEBRAKES**)
RDDR	

Trim_Amount

is a value that varies from -128 to 127 or TO_CURRENT.
A zero value will reset the axis curve so that there's no trim applied.
A positive value increases the trim value, and correspondingly a negative value decreases it.
The TO_CURRENT keyword reads the current values from the axis and sets the trim to those values when the axis is centred.

Ok, so let's look at an example that uses the throttle's Range and Antenna knobs to adjust the trim on the joystick X and Y axes, using Type 1 statements.

```
RNG 1 12 TRIM (JOYX, 20+) TRIM (JOYX, 20-)
ANT 1 12 TRIM (JOYY, 20-) TRIM (JOYY, 20+)
```

Rotating the ANT knob clockwise for example keeps subtracting 20 from the Y axis values, which is equivalent to pushing the joystick forwards. And this would be useful for a plane that was climbing when the joystick was centred.

I could set up Button S2 on the joystick to cancel the effects of the trimming, like this:

```
BTN S2 TRIM (JOYX, 0) TRIM (JOYY, 0)  Rem Remove the trim from both axes
```

or the following line is equivalent to this:

```
BTN S2 TRIM (JOYSTICK, 0)
```

I could also add a specific amount of trim to axes like this:

```
BTN S4 TRIM (JOYX, 5) TRIM (JOYY, -10)
```

Finally I could hold the joystick in a position, and set the trim so that when the joystick is released, it maintains those trim values, like this:

```
BTN S2 /I TRIM (JOYSTICK, TO_CURRENT) Rem Trim to current values
```

```
/O TRIM (JOYSTICK, 0) Rem Cancel any trim
```

There's a slight problem here though. Notice how I said "when the joystick is released." Let me explain further. If you're flying along and you're holding your joystick forwards away from its central position, to keep your aircraft level because it always wants to climb, and then trim the joystick to its current values, what you would probably expect is that you can return the joystick to its central position, and your aircraft keeps flying level. But that isn't going to happen with the above statement unless you immediately release the joystick when you set the trim. Why?

Because the value you want to trim to is calculated assuming that the joystick is in its central position. But it's not. You're pushing it forward at the time the trim is set.

And so as soon as you set the trim, it's going to look like you're actually pushing the joystick a lot further forward, so that when you bring the joystick back into its central position, it appears to the sim that you're holding it forward in the position where you were maintaining level flight. You'll probably need to re-read this last paragraph to grasp what I'm getting at. *I did!*

So how can we overcome this?

Well, we can do it the easy way, or the hard way. Here's the easy way then:

```
BTN S2 HOLDTRIM (JOYSTICK)
```

Now the reason why I'm going to show you the "hard" way soon is that it makes it easier to understand how to use this statement. So the way to use this statement is like this. Hold your joystick in a position where your aircraft is flying level. Now press and keep pressing button S2. Bring your joystick back to its central position, and only then stop pressing button S2. So long as S2 is being pressed whilst you move the joystick, then your plane will continue to fly level, and once the joystick is back in its central position, then you can let go of S2 and put your feet up.

Now let's see the "hard" way of implementing this. Well in fact it's not that hard, and it helps to explain what the above statement does. We need to use the TRIM TO_CURRENT statement in combination with the LOCK and UNLOCK statements (*see later notes*), like this:

```
BTN S2 /P LOCK (JOYSTICK, LASTVALUE) TRIM(JOYSTICK, TO_CURRENT)
       /R UNLOCK (JOYSTICK)
```

Now, whilst I'm pushing my joystick forwards to maintain level flight, if I press **and keep pressing** button S2, the aircraft will maintain level flight, whilst I return the joystick to its central position, and when I'm there, I release button S2. So how does this work. Well the first thing that happens when I press S2 is that the joystick is "locked" to its current values. And straight away as well, the trim is calculated from these locked values. When the joystick is returned to its central position, we unlock the joystick by releasing S2, and the plane flies level because we've trimmed the axes already. And this is effectively what the BTN S2 HOLDTRIM (JOYSTICK) gets translated into by the compiler.

NOTES

1. *A* TRIM *value change results in an integer addition or subtraction to the axis curve, i.e. it just shifts the whole curve in one direction or another. It doesn't matter whether you are using a linear or adjusted response curve.*

2. *A trimmed linear response curve will not allow for the full range of axes.*

3. *Reversing an axis will not alter the direction of the* TRIM *function - it stays the same. Digital statements don't reverse with their analogue counterparts.*

4. *Be careful where you put the + and - signs in a trim statement, as with many axes statements. On the left hand side, they specify the trim amount, on the right of the number the add/subtract to the currently set trim value. See the section titled "*Understanding the Mouse Device and the Microstick*", to understand better the difference between "+ -" signs and their effect when on the left or right hand side of values. The Composer will talk you through this correctly.*

5. *These* HOLDTRIM *statements are all perfectly valid:*

```
BTN T6 a b HOLDTRIM (RNG) c d
BTN S4 /P a HOLDTRIM (RNG)
       /R b
BTN S1 a { HOLDTRIM (JOYY) b HOLDTRIM (ANT) }
```

Note as well that multiple HOLDTRIM *statements as in this last must be grouped in curly brackets.*

6. *You cannot have a macro called* TRIM*, but you can have* Trim *and* Trim_Hold *for example.*

7. *You can use the* AutoRepeat (/A) *slash modifier in conjunction with the* TRIM *statement to control any axis from any button or hat. For example these statements trim the joystick axes.*

```
BTN H1U /A TRIM (JOYY, 5-) DLY(120)
BTN H1D /A TRIM (JOYY, 5+) DLY(120)
BTN H1L /A TRIM (JOYX, 5-) DLY(120)
BTN H1R /A TRIM (JOYX, 5+) DLY(120)
```

In addition, it is also possible to operate axes which aren't even physically present (i.e. rudder or toe brakes) using statements such as these on their respective axes. (See the TM reference book regarding the Apply enable/disable Windows axes states *checkbox on the Cougar Control Panel for further information as to how to report axes present to Windows even if they're not physically present.*)

For example:

```
BTN H4L /A TRIM (RDDR, 5-)
BTN H4R /A TRIM (RDDR, 5+)
```

6.5 DISABLING AXES

All of the analogue axes will be reported to the sim as being present by default, **except** for the Microstick axes. There are some circumstances when it is desirable for them to be disabled, such as when you want to use them to produce only keyboard characters with digital type statements, before the sim starts. Axes can be disabled with configuration statements in the joystick file as follows:

Configuration statement
DISABLE Axis_Identifier

where :

Axis_Identifier is:

THR	
RNG, ANT	(together termed **ROTARIES** – see below)
LBRK, RBRK	(together termed **TOEBRAKES** – see below)
RDDR	

Furthermore, it makes sense to be able as well to define a group of these axis identifiers together, so that they can be disabled with a single statement. For example:

DISABLE ROTARIES	is converted by the compiler into	DISABLE RNG DISABLE ANT
DISABLE TOEBRAKES		DISABLE LBRK DISABLE RBRK

As with all configuration statements, these will appear on their own line in the joystick file, with only a REM statement being a permitted addition.

So, statements

```
DISABLE THR Rem Disable the throttle
DISABLE ANT Rem Disable the antenna knob on the TQS
```

are fine, whereas:

DISABLE (THR, ANT, RNG) *isn't*.

It may look neater but there are advantages from both a programming and user point of view in keeping everything on separate lines (*for one thing it's easier to REM out a single statement*). Note that this new DISABLE statement replaces the USE NO statement found in the original TM syntax (USE NOMOUSE, USE NOTHR) etc. Also, as axes are present by default, you don't need to use statements such as USE RCS, USE TQS etc. as per the previous TM HOTAS.

6.5.1 Disabling and Enabling an axis in flight with LOCK, UNLOCK

So now we know how to disable an axis using a configuration statement. Once the axis is disabled, it cannot be seen by the sim as an analogue axis at all. So it either does nothing, or it can be programmed digitally.

Now there are some occasions when you might for example, want to be able to move an analogue axis to take advantage of the digital type statement programmed onto it, but not change the analogue value from it. Unfortunately, you cannot remove or add axes when in a game – that will usually result in your system locking up, the game crashing or generally any other similarly undesired result.

So if we're not allowed to remove an axis in flight, and add it back, the only way we can get round this is to hold the axis at a certain defined value, and move the axis allowing only its digital output to change. We can do this with the LOCK statement, and enable the axis again with the UNLOCK statement. Here's the syntax:

Command syntax

LOCK (Axis_Identifier, Lock_Value%)

UNLOCK (Axis_Identifier)

where:

Axis_Identifier is:

JOYX, JOYY	(together termed JOYSTICK)
THR	
RNG, ANT	(together termed ROTARIES)
MIX, MIY	(together termed MICROSTICK)
LBRK, RBRK	(together termed TOEBRAKES)
RDDR	

Lock_Value is:

either 0 to 100%, or simply just LASTVALUE
For example:

```
BTN S2 /I LOCK (THR, 100%)
       /O UNLOCK (THR)
```

The LOCK statement is used therefore to force the axis to be seen generating a single value, either something in the region of 0 to 100% of its range, or the last value it was generating with the LASTVALUE syntax. The analogue nature of the axis can be restored using the UNLOCK statement.

Confused? Well the easiest way to understand this is to look at some examples. So here goes:

1. Let's say that we want the RNG knob to be seen purely as an analogue axis, except when Button S3 is held in, when we want it to generate a digital statement, maintaining the last RNG value.

```
RNG /I LOCK (RNG, LASTVALUE) 2 5 a b c d e
    /O UNLOCK (RNG)
```

So when button S3 isn't pressed, the RNG knob is seen as a default analogue axis, and is assigned its action by the game. When Button S3 on the joystick is pressed, the RNG knob now no longer changes its analogue output – it holds the value it was last at, but now it can be turned to generate the "a b c d e" characters as per the Type 2 statement programmed onto the /I statement. I hope that's fairly clear, so let's see just how powerful this is:

2.
```
ANT  /U /I LOCK (ANT, LASTVALUE) 2 10 1 2 3 4 5 6 7 8 9 0
     /O UNLOCK (ANT) Rem Game assigned Axis
     /M UNLOCK (ANT) Rem Game assigned Axis
     /D LOCK (ANT, 0%) 3 Lower_flaps ^ Raise_Flaps
```

With the dogfight switch in the down position (/D), the Antenna knob is going to be used digitally to operate the flaps with a Type 3 statement, and the game will read a zero value from the control it has assigned to the Antenna knob. When the dogfight switch is moved to the middle position (/M), the Antenna knob will behave purely as its assignment by the game. In the dogfight switch's up position (/U), then if Button S3 isn't being pressed, it'll still behave as a default analogue control. When S3 is held in, the last Antenna analogue value will be locked, and the numbers 1 to 0 will be generated when the Antenna knob is rotated.

Using LOCK and UNLOCK, in combination with digital statements, provides a very powerful means of programming any axis. Do be careful though – the potential to get things rather wrong exists here!

NOTES

1. *You cannot disable the Joystick or Microstick axes with the* DISABLE *configuration statement. The joystick axes have to be present - a requirement of DirectX.*

2. *When you disable an axis in a file, the controllers have to go through a sequence whereby they need to report to Windows that an axis is no longer present. This can take a little time, and so downloading such a file, that results in a change in the number of currently reported axes to Windows, will take a little longer.*

3. LOCK *and* UNLOCK *statements must be preceded by* /U, /M, /D, /I, *or* /O *when used in axis statements. So this would generate a compiler error.*

ANT LOCK (RNG, LASTVALUE)

But: BTN S2 LOCK (RNG, LASTVALUE) *is fine!*

6.6 AXIS MAPPING (SWAP)

Axis mapping allows you to swap the axes about, both before taking off, and in-flight. This is all achieved through the SWAP statement.

Configuration statement

USE SWAP (Axis_Identifier, Axis_Identifier)

Command syntax

SWAP (Axis_Identifier, Axis_Identifier)

where:

Axis_Identifier is one of the following:

JOYX, JOYY	(together termed **JOYSTICK**)
THR	
RNG, ANT	(together termed **ROTARIES**)
MIX, MIY	(together termed **MICROSTICK**)
LBRK, RBRK	(together termed **TOEBRAKES**)
RDDR	

For example:

```
USE SWAP (ANT, RNG)
```

results in the Antenna and Range axes on the throttle being swapped around. More examples:

```
BTN S1 SWAP(JOYY, THR)          REM switch the Y and Throttle axes
BTN S2 SWAP(JOYSTICK, MICROSTICK)   REM switch the Joystick X and
                                    Y with Microstick X and Y
```

The last example is converted by the compiler into:

```
BTN S2 SWAP(JOYX, MIX) SWAP(JOYY, MIY)
```

and therefore it's important to note that if you're swapping more than one axis at a time, you can only do so with the same number of axes. So:

```
BTN S2 SWAP(JOYSTICK, MICROSTICK)
```

is fine, *but*

```
BTN S2 SWAP(JOYSTICK, THR)
```

will result in a compiler error, because JOYSTICK defines 2 axes (JOYX and JOYY) but the THR is a single axis.

NOTES

Swapping axis results in their response curves being swapped with them. However any digital statements on those axes do not get swapped over.

6.7 REVERSING THE DIRECTION OF AN AXIS (REVERSE, FORWARD)

It is possible to reverse the default direction of an axis, using the REVERSE statement:

Configuration statement

```
USE REVERSE (Axis_Identifier)
```

Command syntax

```
REVERSE (Axis_Identifier)
FORWARD (Axis_Identifier)
```

where:

Axis_Identifier is one of the following:

JOYX, JOYY	(together termed **JOYSTICK**)
THR	
RNG, ANT	(together termed **ROTARIES**)
MIX, MIY	(together termed **MICROSTICK**)
LBRK, RBRK	(together termed **TOEBRAKES**)
RDDR	

This can be very useful for example in a helicopter sim if you want to use the throttle in the opposite direction compared to a jet, or if you're like me and you like your rudders to work in the opposite direction to a real aircraft!

If I wanted to by default change the direction of my rudders, without needing to press a button, I could do so by placing USE in front of the REVERSE statement to convert it into a configuration statement like this:

```
USE REVERSE (RDDR)
```

This would then appear near the top of my joystick file on its own line. Let's look at another example:

```
BTN H1U   /I REVERSE (JOYY)
          /O FORWARD (JOYY)
```

Pressing HAT 1 Up, with S3 in would result in the joystick's Y axis working in reverse. Pressing HAT 1 Up without button S3 in would return the joystick Y axis to its normal direction. Note that as with other properties of axis statements, the effect of the REVERSE or FORWARD statements stays with the axis if it is mapped or changed elsewhere, but also note that digital statements do not get reversed.

6.8 THE USE AXES_CONFIG STATEMENT

We've discussed in depth how to disable, swap and reverse axes on a per axis basis. It is also possible to set all of this up in a configuration statement in a joystick statement using the USE AXES_CONFIG statement, which the compiler converts into the necessary configuration statements discussed previously. This is a complex statement, and one that will rarely be used, but it can be useful in some circumstances …

Let's look at the syntax first …

Configuration statement:

USE AXES_CONFIG **(***DX-axis1*, HOTASaxis1), (*DX-axis2*, HOTASaxis2) etc.

ie. DX-axis1 is assigned to HOTASaxis1

So what's the difference between a *DX-Axis* and a HOTAS axis? The last one's the easiest to explain so I'll do that one first. A HOTAS axis is any of the 10 physical axes on the Cougar (ie. JOYX, JOYY, THR, RDDR, ANT, RNG, MIX, MIY, LBRK, RBRK). A *DX-Axis* is a little harder to explain…

It is the axis that the Cougar reports to DirectX as being present, which a game then allocates a function to. Now, although we have 10 available axes to use, DirectX 8 only supports 8 axes for USB devices (and DirectX 7 only 6). What we do to make your life easier is to say to Windows, "*Use the Joystick X axis for DX-axis1, Joystick Y axis for DX-axis 2*" etc. It doesn't know whether we have a throttle, RNG knob etc. because it uses these axes even if you're using a steering wheel or other controller. The *DX-axes* are named with the Identifiers below, and are assigned with the Cougar to:

DirectX axis	HOTAS assigned axis	Syntax
X axis	Joystick's X axis	JOYX
Y axis	Joystick's Y axis	JOYY
Z axis	Throttle	THR
Rotation X	Throttle's Antenna knob	ANT
Rotation Y	Rudder's Left Toe brake	LBRK
Rotation Z	Rudder	RDDR
Slider 0	Throttle's Range knob	RNG
Slider 1	Rudder's Right Toe brake	RBRK

Note: The Toe brake assignments may be reversed when the Cougar rudders are produced

Getting back to how we use this statement, there are 4 basic rules to remember:

1. Any axis (DirectX or HOTAS Cougar) not reported will be disabled for analogue.
2. Axes in brackets are mapped to each other.
3. Any HOTAS Cougar axis with a negative sign '-' in front of it will be reversed.
4. You must have axes 1 and 2 present, as Windows requires them to be present.

So let's look at an example:

USE AXES_CONFIG (1, RNG), (2, ANT), (3, -THR)

In this example:

1. The Range knob is assigned to DirectX axis 1, and so will be seen as a joystick X axis.
2. The Antenna knob is assigned to DirectX axis 2, and so will be seen as a joystick Y axis.
3. Throttle axis is assigned to DirectX axis 3, which is normal but the negative sign in front of it means that the throttle is reversed - useful for helicopter sims.
4. No other axes are reported and so are not available to a game. They can on the Cougar be programmed digitally of course.

You can see that with a single statement, it is possible to swap, disable, and reverse axes all in one statement.

NOTES

1. *You cannot use the configuration statements* DISABLE, USE REVERSE *or* USE SWAP *in conjunction with the* USE AXES_CONFIG *statement. Doing so will generate a compiler error. But on button statements, you can still use* SWAP, REVERSE *statements if the axis is present.*

2. *It's much easier to use the* USE PROFILE *statement in conjunction with the Cougar Control Panel! See earlier notes.*

3. *DirectX does its own mapping of axes reported to it, so when using the Cougar Control Panel, you may find that the results on your Cougar axes don't match what you'd expect. You'll just need to experiment to get the required results, I'm afraid.*

7. Mouse Programming

7.1 UNDERSTANDING THE MOUSE DEVICE AND THE MICROSTICK

In the final part of this chapter, we're going to look at how the mouse device works and show you a few clever things we can do with it. But I'm not going to go straight into the associated statements and syntax, as we've done before, because we actually need to understand a little as to how a mouse works.

I'm sure by now you've used the microstick on the TQS throttle and thought of it as a mouse controller, because it moves the mouse. However, it is important to realise this point:

The Microstick is NOT a mouse.

The microstick is like a little mini-joystick. It *usually* controls the mouse, because the compiler tells it to. The compiler assigns to its two axes, (MIX and MIY) the **mouse device**, without you being aware of this.

So what's a mouse device then? And why can't we just assign the mouse X and Y axes to the microstick's X and Y axes?

The reason is this: the mouse *as a device* **does not** consist of fixed axes as such. Confused? Well, if you think about a joystick, when you move it around, it generates fixed X and Y coordinates for itself. If the joystick was controlling a cursor as it does in Foxy's Joystick Analyser, its movement would correspond to the movement of the joystick. If the joystick stopped moving, the cursor would stop moving too. But the microstick programmed as a mouse behaves differently. If you move that like the joystick, and hold it in a position away from its centre, the mouse keeps moving. It doesn't stop even though the microstick is stationary. This is because the microstick isn't telling the computer "*Hey, move the mouse to this X, Y position and then stop,*" rather it's saying "*Keep moving the mouse at a speed of 3 along its X axis, and a speed of 2 along its Y axis until I instruct you differently.*" Then, when the microstick is returned to the centre position, the cursor doesn't move to the centre of the screen; it stops where it is, because the microstick has changed its instructions to "*OK, you can stop moving the mouse along its X and Y axes now*."

So a mouse cursor can be made to move by assigning non zero values to MouseX and/or MouseY (MSX and MSY). And if we instruct the microstick, to change the values of MSX and MSY, then the microstick will control the mouse.

And the reason why we've implemented this in this way, should I hope now be starting to dawn upon you. We can use anything to adjust MSX and MSY. The Microstick, the joystick, a hat, a button, logical flags …. you name it. And they can do so at the *same* time. So we could program the microstick for quick movement of the mouse, and a hat for fine movement!

7.2 USE MTYPE - THE SIMPLEST WAY OF ASSIGNING THE MOUSE TO THE MICROSTICK

Later on in this chapter, I'm going to discuss how you can setup the mouse onto the microstick, to get exactly the response you want from it. But to do this, requires a detailed understanding of using Digital Type statements using MSX

and MSY statements, and it's not for the faint hearted. However, thankfully there are a couple of statements that we can use to assign the mouse to the microstick very easily. These are the USE MTYPE and USE MICROSTICK AS MOUSE statements. Let's start off with the USE MTYPE statement as that's very easy to understand and use.

Configuration statement:

USE MTYPE Type **-** REVERSE_*type*

where:

Type: is A1 to A5 and describes which buttons on the throttle will be used for the left and right mouse buttons as follows:

Type	Left mouse button	Right mouse button
A1	T1	T6
A2	T6	T1
A3	T1	none
A4	T6	none
A5	none	none

T1 is the button built into the microstick - the microstick depresses to activate it, and T6 is the button on the Range knob.

REVERSE_*type* is REVERSE_UD and/or REVERSE_LR

The REVERSE_UD reverses the Up and Down (Y) mouse axis direction, and REVERSE_LR reverses the Left and Right (X) directions.

Examples: USE MTYPE A3

assigns the mouse to the microstick, and the mouse left button to T1.

USE MTYPE A5 - REVERSE_UD

assigns the mouse to the microstick, reversing the direction of its Y axis, and doesn't assign any mouse buttons.

NOTES

1. *The mouse response is set up by the compiler to give a pre-defined mouse response, which should be an acceptable response for resolutions up to 1024 by 768. You cannot alter the mouse response with a* USE MTYPE *statement, so if you're running at a higher resolution, or want a more/less responsive*

mouse, then you need to use the USE MICROSTICK AS MOUSE *statement that we're coming onto in the next section.*

2. *The microstick as we've said before is an analogue controller. It isn't a 4 button controller, as per the original TM HOTAS, and therefore buttons T11 to T14 no longer exist for programming. You can emulate T11 to T14 with appropriate* Type *statements if you so wish - see the help topic in Foxy's help file titled: " Converting TQS T11 - T14 statements for use with the Cougar's microstick.". Remember that the Microstick is just that – a controller with 2 axes, not a series of buttons. It's far more powerful that way.*

3. *If you assign any curves to the Microstick, they will not affect the mouse, as it is assigned as a digital statement, and digital statements are not affected by analogue curves.*

4. *If you use a* USE MTYPE *statement in your joystick file, that sets up any mouse button on T1 or T6, then you cannot program those positions. So let's say that we have:*

USE MTYPE A3

which the compiler, invisible to you, also sets up the statement:

BTN T1 /H MOUSE_LB

Now, if you elsewhere in your file have:

BTN T1 *somemacro* or BTN T1 /H MOUSE_RB

then the compiler will generate an error. If you want to use the USE MTYPE *statement to setup the mouse onto the microstick, but want to program T1 and/or T6 separately, then use a* USE MTYPE A5 *or appropriate statement that doesn't assign any mouse button to the throttle button you wish to program.*

7.3 USE MICROSTICK AS MOUSE

The second way of assigning the mouse to the microstick is with the USE MICROSTICK AS MOUSE statement. This has the advantage over the USE MTYPE statement in that you can change the default behaviour of the mouse, ie. how fast it moves. It also assigns the left mouse button to T1 on the microstick by default, although you can turn this off with the - NO_BUTTON modifier. For most flight sims of course, we want the left mouse button assigned to T1. The USE MICROSTICK AS MOUSE statement effectively instructs the compiler to set up either digital Type 6 statements for the mouse on the microstick, or if a starting value is provided, Type 5 statements.

Let's take a look at the syntax, then:

Configuration statements

For Type 6 statements: *(No starting value provided)*

USE MICROSTICK AS MOUSE (Scale value, Increment value) - *Modifier*

For Type 5 statements: *(Starting value provided)*
USE MICROSTICK AS MOUSE (Scale value, Increment value, Starting value) - *Modifier*

*(**Note**: USE MICROSTICK AS MOUSE ( ) statements also assign the left mouse button to button T1 on the microstick, unless a - NO_BUTTON modifier is present.)*

where:

Scale value: A number between 2 and 12. This affects how many bands the microstick axes are divided up into.

Increment value: Incremental value for each band, a number between 1 and 63.
(Note that the Scale value multiplied by the Increment value must be less than 128. Don't even think about asking why!)

Starting value: The starting value for a Type 5 statement, from which to apply the incremental value to. This is the initial speed the mouse will move at when the microstick is moved away from its centre position.

Modifier:

REVERSE_UD:	Reverses the Microstick's Y axis.
REVERSE_LR:	Reverses the Microstick's X axis.
NO_BUTTON:	Instructs the compiler not to set up button T1 as the left mouse button. This allows you to program button T1 with BTN T1 statements.

Let's dive straight into some examples.

```
USE MICROSTICK AS MOUSE (12, 2)
```

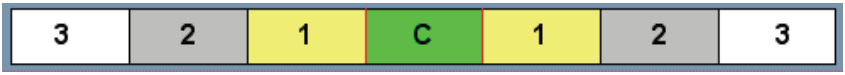
This configuration statement would assign the mouse to the microstick, and setup T1 as the left mouse button. Take a look at these two statements:

```
USE MICROSTICK AS MOUSE (6, 4)
USE MICROSTICK AS MOUSE (7, 3, 2)
```

These too would also assign the mouse to the microstick, and set up T1 as the left mouse button. So here we have 3 statements, that are clearly different, but I've just said that they all do the same. Well, I'm correct in saying that they all do the same *in terms of* assigning the mouse to the microstick, but they differ in the mouse response you'll see on the microstick. Which means of course that I need to try to explain what the numbers in the brackets do.

Well we're going to cover *exactly* what they do in the next section, which makes for some pretty tough reading. So here's a less heavy explanation which suits my simple brain!

Let's consider the Microstick's X axis only. What any USE MICROSTICK AS MOUSE statement does is to split the axis up into a series of bands or regions, as in the diagram below. (*For sake of argument, I've made all these bands the same size, although that's not actually the case with this statement for the purists amongst you.*)



Now the green band represents the centre of the axis, ie. the Microstick is in its untouched centre position. Here of course we don't want the mouse to move. Either side of centre, are two yellow bands (1), which is where we want the mouse to start moving when the microstick is moved into these bands. And there are a couple more bands (2 & 3) that the microstick will move into as it is moved further away from its centre position.

Let's go back to our syntax:

USE MICROSTICK AS MOUSE (Scale value, Increment value, *optional* Starting value)

The Scale value determines how many bands the microstick axis is split up into. The value for the Scale value isn't the same as the number of bands, it's actually part of an equation, but basically the larger the Scale value, the greater the number of bands the axis is split into.

The Increment value determines how much faster to move the mouse as the microstick is moved into each band. Let me explain with an example, illustrating what happens when the microstick is moved from its centre position to its extreme position, as it moves through these bands. I'll pick up on the Starting value later.

USE MICROSTICK AS MOUSE (4, 2)

Band C: The microstick is in its centre position, and so the mouse doesn't move.
Band 1: The mouse now starts moving at a rate given by the Increment value, a "*speed of 2*" if you like.
Band 2: The rate of the mouse is increased by the Increment value, so it now moves at a *speed of 4*.
Band 3: The rate of the mouse is increased by the Increment value, so it now moves at a *speed of 6*.
Band 4, 5, 6 etc.
Depending on how many bands the Scale value has created, you can see that the mouse moves faster and faster as the microstick is moved through each successive band. Moving back through the bands towards the centre position will of course decrease the mouse rate as well.

Let's now consider the effect of providing a starting value.

```
USE MICROSTICK AS MOUSE (4, 2, 1)
```

The Starting value determines the rate at which the mouse will move in band 1. After that, it's exactly the same with the mouse speed increasing by the Increment value as the microstick is moved though the rest of the bands. So:

Band C: The microstick is in its centre position, and so the mouse doesn't move.
Band 1: The mouse now starts moving at the rate given by the Starting value, a "*speed of 1*" if you like.
Band 2: The rate of the mouse is increased by the Increment value, so it now moves at a *speed of 3 (ie. 1 + 2, the Starting value + Increment value)*.
Band 3: The rate of the mouse is increased by the Increment value, so it now moves at a *speed of 5*.
Band 4, 5, 6 etc.
Depending on how many bands the Scale value has created, you can see that the mouse moves faster and faster as the microstick is moved through each successive band. Moving back through the bands towards the centre position will of course decrease the mouse rate as well.

So don't worry if you don't take all of this in at first, I'm just trying to get across a general message: Bigger numbers means a faster mouse in general.

We can also reverse the direction that the mouse moves in with the microstick, as we've done with other statements we've looked at in previous chapters, like this:

```
USE MICROSTICK AS MOUSE (7, 3, 2) - REVERSE_UD
USE MICROSTICK AS MOUSE (7, 3, 2) - REVERSE_LR
```

Before I leave this section, I should for the sake of completeness cover the other uses for this statement, and that is assigning the mouse to other axes.

7.3.1 Assigning other axes to mouse axes

Now, the USE MICROSTICK AS MOUSE statement is in fact, just a special case of the following statement (*special because it also assigns the left mouse button to T1*):

Configuration statements

For Type 6 statements:

USE *Axis_Identifier* AS *Mouse_Axis* (Scale value, Increment value) - REVERSE_*type*

For Type 5 statements:

USE *Axis_Identifier* AS *Mouse_Axis* (Scale value, Increment value, Starting value) - REVERSE_*type*

where:

Axis_Identifier is one of the following:

JOYX, JOYY	(together termed **JOYSTICK**)
THR	
RNG, ANT	(together termed **ROTARIES**)
MIX, MIY	(together termed **MICROSTICK**)
LBRK, RBRK	(together termed **TOEBRAKES**)
RDDR	

Mouse_Axis is one of the following:

MOUSE
MOUSEX
MOUSEY
MOUSEZ (the wheel on a mouse)

Scale value: A number between 2 and 12. This affects how many bands the *Axis_Identifier* will be split into.

Increment value: Incremental value for each band, a number between 1 and 63.
(Note that the Scale value multiplied by the Increment value must be less than 128. Don't even think about asking why!)

Starting value: The starting value for a Type 5 statement, from which to apply the Incremental value to. This is the initial speed the mouse will move at when the *Axis_Identifier* is moved away from its centre position.

REVERSE_*type*: reverses an axis direction and is either:

REVERSE_UD:	when the *Axis_Identifier* consists of 2 axes (JOYSTICK, ROTARIES, MICROSTICK, TOEBRAKES).
REVERSE_LR:	when the *Axis_Identifier* consists of 2 axes this can be used on its own or in conjunction with REVERSE_UD.
REVERSE_DIR:	reverses a single axis. Cannot be used with REVERSE_UD, or REVERSE_LR.

So we can use other axes to control mouse axes, in exactly the same way. Here are some examples then:

```
USE ROTARIES AS MOUSE (6, 4)
USE JOYSTICK AS MOUSE (11, 2, 0)
USE ANT AS MOUSEY (5, 2) - REVERSE_DIR
USE JOYY AS MOUSEZ (9, 3)
```

This last one controls the mouse wheel using the joystick Y axis!

In summary then we've looked at two statements that can be used to set up the mouse onto the microstick, and in the last section, onto other axes. Remember as well that when we looked at HAT programming, we could also assign the mouse to a hat using the statement:

```
USE HAT1 AS MOUSE (2)
```

for example. In fact we can even have the mouse on the microstick, and on the hat at the same time, using the microstick for getting the mouse to the required position quickly, and the hat for fine adjustments of that position! *[Insert applause here]*

I want to now explain exactly what is going on, or I should say how the compiler interprets your statements and what it creates in the way of digital type statements for the microstick axes. This is pretty tricky to explain, and hence the reason why I didn't jump into it straight away. This next section is for advanced users only (*believe me it took me ages to get my head round this!*). But if you do get your head round this, then you can create your own custom mouse devices on any axis, to get exactly the response you want. You will also then be able to mix mouse statements with other digital statements on the microstick, so for example with button S3 out you could have the microstick controlling your targeting cursor, and with S3 in, controlling your mouse. Clever huh!

7.4 CREATING A CUSTOM MOUSE ON THE MICROSTICK

We've seen how we can assign the mouse to the microstick in the previous sections with the USE MTYPE and USE MICROSTICK AS MOUSE statements. What these statements do quite simply, is to program the microstick axes with digital type statements. In this section then, I want to explain how you can create your own digital statements to program the mouse onto the microstick, or elsewhere.

You can program the microstick with any digital statements, because they are after all just axes subject to the same rules as other axes. I'll start off demonstrating the use Type 1 and Type 2 statements, although there's no reason why we shouldn't use any of the 6 digital type statements. Let's take a look at a some appropriate Type 1 digital statements:

```
MIX 1 14 MSX (2+) MSX (2-) MSX (0)
MIY 1 14 MSY (2-) MSY (2+) MSY (0)
```

Well, these look like ordinary Type 1 statements, except that we have some "-" and "+" signs in them. To understand these, I'll walk you through what happens when we move the microstick. Let's remind ourselves as to how a Type 1 digital statement works. If we had the following:

```
MIX 1 6 r l c
```

then the microstick X axis, if moved from its extreme left, to its extreme right, and then all the way back again to its extreme left, would produce the characters:

r r r c r r r l l l c l l l

Now in our example, we have:

```
MIX 1 14 MSX (2+) MSX (2-) MSX (0)
```

We're not producing now a series of characters. The numbers in the brackets specify what to add or subtract to the current mouse buffer - in other words, how much faster, or slower, should the mouse move.

Let's say the mouse is stationary, and the microstick is in its centre position. The centre position of the microstick's axis is given by the centre "character" part of the digital axis statement - ie. MSX (0). This instructs the mouse device to have zero speed, ie. to be stationary along its X axis. As we move the microstick to the right, the mouse will start to move to the right at a "rate of 2", as a positive 2 is added to the mouse buffer. Move the microstick more to the right, and it'll move the mouse at a rate of 4 …. move it all the way to the right, and it'll move it right at a rate of 14 (7 x 2). Let go of the microstick and the mouse's rate of movement to the right slows down as successive amounts of rate are subtracted from the

mouse buffer, and hence mouse speed, until the stick is at the centre. It stops moving here because once again, we're at the centre "character" of the statement, ie. MSX (0).

The same occurs for exactly the same reason when the microstick is moved up and down.

Notice that if I wanted the microstick Y axis to move the mouse in the same way that a joystick would (pull back and the mouse goes up), then I'd need to change the statement to:

MIY 1 14 MSY (2+) MSY (2-) MSY (0)

This is because if you increase the Mouse Y buffer (MSY) the mouse moves down the screen, and vice versa. An important syntax point to note therefore is the inclusion of "**+**" or "**-**" characters *within* the brackets, and the fact that they appear **after** the number. They indicate that the value before them is either to be added (+) or subtracted (-) from that mouse axis's buffer. You'll understand this better when we look at Type 2 statements now. I hope I haven't lost you. *Yet!*

I could also use Type 2 statements to assign the mouse to the microstick axes:

MIX 2 9 MSX(-8) MSX(-4) MSX(-2) MSX(-1) MSX(0) MSX(1) MSX(2) MSX(4) MSX(8)
MIY 2 9 MSY(8) MSY(4) MSY(2) MSY(1) MSY(0) MSY(-1) MSY(-2) MSY(-4) MSY(-8)

These are standard type 2 digital statements. They divide up each axis into 9 equal bands, and assign the actual mouse buffer values into each band. So moving the microstick right along its X axis, results in it initially moving at a rate of 1, and then moving the microstick further it'll move at a rate of 2 … then 4 … and finally 8. Notice here the position of the "-" sign. It is **before** the number this time, which means it doesn't get subtracted from the mouse buffer. Rather, when the microstick moves into this area, it takes up that exact negative value.

In terms of syntax, note that the following statements are the same:

MIX 2 7 MSX(-4) MSX(-2) MSX(-1) MSX(0) MSX(1) MSX(2) MSX(4)
MIX 2 7 MSX(-4) MSX(-2) MSX(-1) MSX(0) MSX(+1) MSX(+2) MSX(+4)

i.e., if a "+" sign is missing, it is assumed to be there on the left of the value.

Whereas:

MIX 2 7 MSX(-4) MSX(-2) MSX(-1) MSX(0) MSX(1) MSX(2) MSX(4)
MIX 2 7 MSX(4-) MSX(2-) MSX(1-) MSX(0) MSX(1+) MSX(2+) MSX(4+)

would produce *very* different results.

With this all in mind, let's see what the Compiler does when it sees a USE MICROSTICK AS MOUSE statement in your joystick file. The Compiler converts these statements into Type 5 and Type 6 digital statements, which are effectively the same as Type 2 and Type 1 statements respectively, except that we can determine the sizes of the bands. Let's recall the syntax the for this statement:

USE MICROSTICK AS MOUSE (Scale value, Increment value, *optional* Starting value)

I said before that the Scale value is used to determine the number of bands to divide the axis into. It does this via the formula:

Number of bands = (Scale value x 2) - 1

The Compiler then creates these bands with different sizes, using a complex formula I won't go into, and creates Type 6 digital statements if no Starting value is provided, or Type 5 statements if a Starting value is provided.

Type 6 Digital Statements

```
USE MICROSTICK AS MOUSE (2, 2)

MIX 6 3 (2 24 75 98) MSX(2+) MSX(2-) ^
MIY 6 3 (2 24 75 98) MSY(2-) MSY(2+) ^
BTN T1 /H MOUSE_LB Rem Hold down the left mouse button when T1 is pressed
```

Now, we've done something a little different here. I don't have MSX(0) and MSY(0) as the centre characters, like this:

```
MIX 6 3 (2 24 75 98) MSX(2+) MSX(2-) MSX (0)
MIY 6 3 (2 24 75 98) MSY(2-) MSY(2+) MSY(0)
```

Let me explain why as there are advantages and disadvantages in having null characters (^) instead of MSX (0), MSY (0). These type 6 statements differ from Type 5 statements in that they add or subtract to the X and Y values in the mouse buffer, whereas Type 5 statements set the actual values for the mouse buffer. When we use MSX(0) and MSY(0) as centre characters, these reset the mouse buffer to zero so that the mouse is guaranteed to stop moving at the centre position of the axis controlling it. That's a good thing in general. But the beauty of using null characters as centre characters is that we can then assign or control the mouse from hats and buttons as well as the microstick, and ensure predictable results. So I could have:

```
USE MICROSTICK AS MOUSE (2, 2)
BTN H1L MSX(1-)
BTN H1R MSX(1+)
```

and use the microstick for general control of the mouse and Hat 1 left and right as fine control of the mouse's X direction. That's worth remembering … it all depends on how you want your mouse to behave. Unfortunately there's no way to tell the compiler to use MSX(0), MSY(0) as centre characters with a USE MICROSTICK AS MOUSE statement. If you want that, then you need to use the actual MIX and MIY statements or provide a Starting value, which will then set up Type 5 statements that do use MSX(0), MSY(0) as centre characters. Let's see what the Compiler does then with some more USE MICROSTICK AS MOUSE Type 6 statements.

USE MICROSTICK AS MOUSE (3, 4)

is converted into:

```
MIX 6 5 (2 16 32 68 84 98) MSX(4+) MSX(4-) ^
MIY 6 5 (2 16 32 68 84 98) MSY(4-) MSY(4+) ^
BTN T1 /H MOUSE_LB
```

USE MICROSTICK AS MOUSE (4, 2)

is converted into:

```
MIX 6 7 (2 12 23 36 65 78 89 98) MSX(2+) MSX(2-) ^
MIY 6 7 (2 12 23 36 65 78 89 98) MSY(2-) MSY(2+) ^
BTN T1 /H MOUSE_LB
```

USE MICROSTICK AS MOUSE (12, 3)

is converted into:

```
MIX 6 23 (2 4 6 8 11 14 17 21 25 30 36 43 58 65 71 76 80 84 87 90 93 95 97 98) MSX(3+) MSX(3-) ^
MIY 6 23 (2 4 6 8 11 14 17 21 25 30 36 43 58 65 71 76 80 84 87 90 93 95 97 98) MSY(3-) MSY(3+) ^
BTN T1 /H MOUSE_LB
```

USE MICROSTICK AS MOUSE (4, 2) - REVERSE_UD

is converted into:

```
MIX 6 7 (2 12 23 36 65 78 89 98) MSX(2+) MSX(2-) ^
MIY 6 7 (2 12 23 36 65 78 89 98) MSY(2+) MSY(2-) ^
BTN T1 /H MOUSE_LB
```

Now, let's take a look at what happens when we provide a Starting value in the USE MICROSTICK AS MOUSE statement. The Compiler converts these into Digital Type 5 statements.

Type 5 Digital Statements

USE MICROSTICK AS MOUSE (2, 2, 3)

is converted into:

```
MIX 5 3 (0 24 75 100) MSX(-3) MSX(0) MSX(3)
MIY 5 3 (0 24 75 100) MSY(3) MSY(0) MSY(-3)
BTN T1 /H MOUSE_LB
```

Notice that the number of bands that are created is 3. The centre band is used to ensure that the mouse is stationary, and the 2 bands either side of it take up the Starting value, so effectively the Increment value is ignored in this case. Now compare this statement with the next one below:

USE MICROSTICK AS MOUSE (3, 2, 3)

is converted into:

```
MIX 5 5 (0 14 31 69 86 100) MSX(-5) MSX(-3) MSX(0) MSX(3) MSX(5)
MIY 5 5 (0 14 31 69 86 100) MSY(5) MSY(3) MSY(0) MSY(-3) MSY(-5)
BTN T1 /H MOUSE_LB
```

Now you can see the relationship between the Starting value and the Increment value. Or if you can't compare the statement below and you should have it by then.

USE MICROSTICK AS MOUSE (3, 8, 1)

is converted into:

```
MIX 5 5 (0 14 31 69 86 100) MSX(-9) MSX(-1) MSX(0) MSX(1) MSX(9)
MIY 5 5 (0 14 31 69 86 100) MSY(9) MSY(1) MSY(0) MSY(-1) MSY(-9)
BTN T1 /H MOUSE_LB
```

USE MICROSTICK AS MOUSE (12, 2, 3)

is converted into:

```
MIX 5 23 (0 2 4 6 9 12 16 20 25 30 36 43 59 66 72 77 82 86 90 93 96 98 99 100)
            MSX(-23) MSX(-21) … MSX(-3) MSX(0) MSX(3) … MSX(21) MSX(23)
MIY 5 23 (0 2 4 6 9 12 16 20 25 30 36 43 59 66 72 77 82 86 90 93 96 98 99 100)
            MSY(23) MSY(21) … MSY(3) MSY(0) MSY(-3) … MSY(-21) MSY(-23)
BTN T1 /H MOUSE_LB
```

USE MICROSTICK AS MOUSE (3, 8, 1) - REVERSE_UD, REVERSE_LR

is converted into:

```
MIX 5 5 (0 14 31 69 86 100) MSX(9) MSX(1) MSX(0) MSX(-1) MSX(-9)
MIY 5 5 (0 14 31 69 86 100) MSY(-9) MSY(-1) MSY(0) MSY(1) MSY(9)
BTN T1 /H MOUSE_LB
```

And for sake of completeness, the assignment to other axes:

USE JOYSTICK AS MOUSE (3, 2, 3)

```
JOYX 5 5 (0 14 31 69 86 100) MSX(-5) MSX(-3) MSX(0) MSX(3) MSX(5)
JOYY 5 5 (0 14 31 69 86 100) MSY(5) MSY(3) MSY(0) MSY(-3) MSY(-5)
```

USE JOYY AS MOUSEZ (4,2)

is converted into:

```
JOYY 6 7 (2 12 23 36 65 78 89 98) MSY(2-) MSY(2+) ^
```

Well I hope I've managed to explain that clearly! Let's move on.

7.5 USE ZERO_MOUSE

This configuration statement is useful when setting up your own mouse statements on the microstick, in conjunction with /I and /O modifiers, to prevent the mouse from getting stuck.

We've seen in the previous section how we can use digital statements to create a custom mouse on the microstick. A USE ZERO_MOUSE configuration statement can be used to ensure that you don't get a stuck mouse when combining mouse statements with /I and /O modifiers. Consider this example:

```
MIX    /I 1 6 MSX(2+) MSX(2-)
       /O 1 6 RARROW LARROW
MIY    /I 1 6 MSY(2-) MSY(2+)
       /O 1 6 UARROW DARROW
```

In this example, if you hold button S3 in on your joystick, then the microstick will move the mouse. If whilst the mouse is moving you release button S3, the mouse will continue to move, and it will get stuck when it reaches one border of your screen. Inserting a USE ZERO_MOUSE statement will prevent this from happening, forcing the mouse to stop moving when button S3 is pressed/released.

It is worth pointing out here that this stuck mouse behaviour isn't a bug as such. The control of the mouse by the microstick is behaving as it has been programmed to behave. You can always prevent a stuck mouse by returning the microstick to its central position before releasing S3 with the above example. The golden rule here is this when it comes to stuck keys or stuck mouse movements: you must use your controllers' buttons, hats and axes as they have been designed to be used through their programming. This statement is of course a great work around for those of us who never follow this rule :)

7.6 PROGRAMMING WITH MOUSE BUTTONS

You can assign the mouse buttons in buttons statements, axis statements … well in all manner of statements. Here's the syntax then for them:

Mouse button syntax:

MOUSE_LB
MOUSE_RB
MOUSE_MB

(MB = *middle button on a 3 button mouse*)

Here's an example:

```
BTN T1 /I /H MOUSE_RB
       /O /H MOUSE_LB
```

With this statement, we're assigning the left mouse button to the microstick's T1 button when button S3 on the joystick isn't being pressed, and the right mouse button to the microstick's T1 button when button S3 is being pressed.

You can also use KD and KU with the mouse buttons:

```
BTN S1 KD(MOUSE_LB) DLY(2000) KU (MOUSE_LB)
```

When button S1 is pressed, the left mouse button is pressed for 2 seconds, and then released.

7.7 DISABLING THE DEFAULT ASSIGNMENT OF THE MOUSE TO THE MICROSTICK

In the Preferences Window in Foxy, there is a tab titled "Defaults." The purpose of these settings is to instruct the compiler to setup the selected settings, if no statements to the contrary exist in a file.

One of the settings is "Assign mouse to microstick." If this is selected, then when you download a file to your controllers, the compiler sets up the mouse onto the microstick, and the left mouse button onto T1. This is designed like this as most users will want the microstick to control the mouse by default, but may not have got this far into the reference book to understand what statements to use to do this!

If you wanted to be able to use the microstick axes assignable within a game, but not have the mouse assigned to them by default, without having to deselect the default option in Foxy, then you should use the configuration statement:

Configuration statement

```
DISABLE MOUSE
```

in your joystick file. This will prevent the compiler from assigning the default digital statements to the microstick *if you've setup Foxy's preferences to force the compiler to do this*.

You can still assign the mouse to a hat, buttons or other axes. Remember that the mouse device is always present - it's just that you have to assign it to something if you want to control the mouse.

7.8 ADVANCED MOUSE MOVEMENT STATEMENTS

We've already seen that it is possible to move the mouse through programming statements. In the final part of this chapter, we're going to look at how to perform more complex mouse movements.

7.8.1 Defining the screen resolution

With all of the statements that we will be explaining soon, it is **essential** to define the screen resolution that you are using for your game, with the configuration statement:

Configuration statement:

```
USE SCREEN_RESOLUTION (X,Y)
```

Example:

USE SCREEN_RESOLUTION (800,600)

The lowest values you can enter are 640, 480 for X and Y respectively.

Technical note: The 800 and 600 describe the screen resolution in pixels. A pixel is the smallest point you can draw on a screen. So in this instance the screen consists of 800 lines and 600 columns of points. Obviously at a screen resolution of 1600 by 1200, your screen resolution is higher, meaning that images tend to be less jagged.

Then we can use the following statements with the mouse:

Command syntax:

Moving to a specific screen position
MOUSEXY (*Origin*,X,Y)

Moving relative to the current mouse position
MOUSEMOVE (X,Y)

Rotational/Polygon movement
MOUSEROTATE (*Origin*, CentrePoint, Radius, Start angle, Macro1, Rotate direction, Final angle, Number of steps, Macro2)

7.8.2 Moving to a specific screen position

Command syntax:

MOUSEXY (*Origin*,X,Y)
MOUSEXY (*Origin*,X%,Y%)

where

- *Origin* is one of the corner positions on the screen UL, DL, UR, DR.
- X,Y is the X, Y coordinate on the screen where you want to move the mouse to.
- X%, Y% is a percentage movement from 0 to 100% (3 decimal place accuracy) to move the mouse by. Using percentages allows you to change screen resolution without needing to change statement values.

There is no way within a game to know where the mouse is at any one time, or be able to track it. And therefore to be able to move to a specific position, we have to first move the mouse to a corner position on the screen, as we know exactly what the coordinates of this position is from the SCREEN_RESOLUTION statement,

and the compiler can then calculate where to move the mouse to from there. This happens very quickly so it shouldn't be a problem, but you should ensure a mouse button isn't being pressed before doing this.

So the statement:

```
BTN H3D MOUSEXY (UL, 400, 300)
```

will first move the mouse very quickly to the top left corner of the screen, then move it to point 400, 300, ie. the centre of the screen when it is at a resolution of 800 by 600. So let's say that you need to press F2 to get a cockpit view, then need to move the mouse to a button in your cockpit, press the button, and return the view to the forward head up position pressing F1. This statement will do this:

```
BTN H3D F2 MOUSEXY (UL, 400, 300) MOUSE_LB F1
```

Now if we changed the statement to this:

```
BTN H3D F2 MOUSEXY (UL, 50%, 50%) MOUSE_LB F1
```

then if a user changed their resolution for their game to 1600 by 1200, and changed their USE SCREEN_RESOLUTION statement to reflect this, then the statement will work just as it did at 800 by 600.

NOTES

1. *You must have a* USE SCREEN_RESOLUTION ( ) *configuration statement present to use this statement.*

2. *All mouse statements are subject to* RATE *value speed restrictions. If the mouse movement is too slow then reduce the* RATE *statement value. Note that by default if you don't have a* USE RATE (*nnnn*) *statement in your file, it will be set to zero - the fastest response anyway.*

7.8.3 Moving the mouse relative to its current position

Command syntax:

```
MOUSEMOVE (X,Y)
MOUSEMOVE (X%,Y%)
```

where:

- X,Y is the number of pixels you want to move the mouse to relative to its current position.
- X%, Y% is a percentage movement from 0 to 100% (3 decimal place accuracy) to move the mouse by. Using percentages allows you to change screen resolution without needing to change statement values.

Generally then you'll know where the mouse is because you'll use a MOUSEXY statement to position it first.

Let's look at some examples using the MOUSEMOVE statement:

BTN TG1 MOUSEMOVE (20, 50)

Pressing the trigger will move the mouse 20 pixels to the right along the horizontal axis of the screen, and 50 pixels down the vertical axis of the screen.

BTN S1 MOUSEMOVE (20%, 50%)

Pressing button S1 will move the mouse 20% of the screen width to the right along the horizontal axis of the screen, and 50% of the screen height down the vertical axis of the screen. Now if you're running at 800 by 600, this means that the mouse will move 160 pixels to the right (20% of 800) and 300 pixels down (50% of 600).

BTN H2R MOUSEMOVE (30, 0)

Pressing Hat 2 right will move the mouse 30 pixels to the right only … i.e. along a horizontal line. If you wanted to move the mouse left then you could place a '-' sign before the 30.

BTN H2U MOUSEMOVE (0%, -50%)

Pressing Hat 2 up will move the mouse up the screen along a vertical line by 50% of the screen height - equivalent to 300 pixels at a screen resolution of 800 by 600.

NOTES

1. *You must have a* USE SCREEN_RESOLUTION ( ) *configuration statement present to use this statement, or a compiler error will occur.*

2. *All mouse statements are subject to* RATE *value speed restrictions. If the mouse movement is too slow then reduce the* RATE *statement value. Note that by default if you don't have a* USE RATE (*nnnn*) *statement in your file, it will be set to zero - the fastest response anyway.*

3. *You cannot mix % and absolute values in these statements. So:*

BTN H2U MOUSEMOVE (0%, -50%) *is correct, whereas:*

BTN H2U MOUSEMOVE (0, -50%)

will generate a compiler error.

4. *With a* MOUSEMOVE *statement, the mouse moves along both axes at the same time. So a statement such as:*

BTN S1 MOUSEMOVE (100, 100)

will move the mouse diagonally down and to the right, and not 100 pixels along the screen's X axis first, and then 100 pixels down the Y axis.

7.8.4 Rotational/Polygon movement

Command syntax:

MOUSEROTATE **(**Origin, CentrePoint, Radius, Start angle, [Macro], Rotate direction, Final angle, Number of steps **)**

Where:

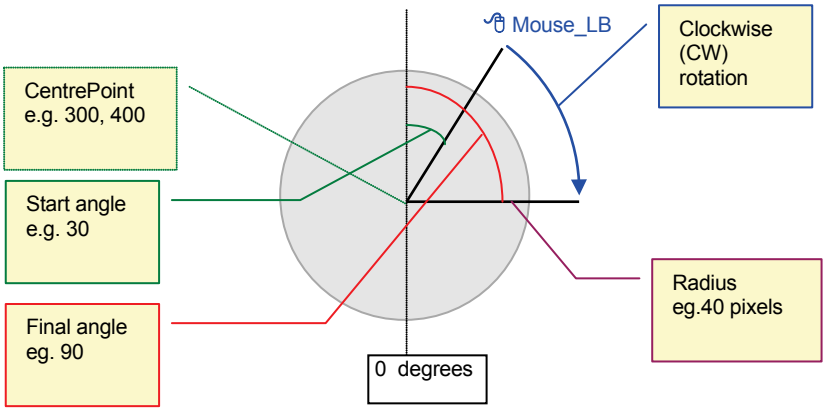
- *Origin* is one of the corner positions on the screen UL, DL, UR, DR.
- *CentrePoint* is the centre of rotation as a coordinate: X,Y or as a %
- *Radius* is the radius of a circle for defining the rotation arc in pixels or percentage (*but see the notes section*)
- *Start angle* is from 0 to 360°
- *Macro* is a simple macro … typically a mouse button press (*but see the restrictions in the Notes section*.) Use a null character if you don't want to insert a macro. The macro must be enclosed by [ ] brackets.
- *Rotate direction* is CW or CCW (Clockwise or CounterClockwise)
- *Final angle* is the amount of rotation you want as an angle between 0 to 1800° (*5 complete revolutions*)
- *Number of steps* defines how smooth the rotation occurs. Values allowed are 1 or 2. These produce movement in the following shapes:

  Number of steps = 1: a square (*err.. 4 sides!*),
  Number of steps = 2: an octagon (*8 sides*)

The larger the step value, the slower the movement, although it's not that dramatic.
All numerical values are accurate to one decimal place (eg. 244.3, 301.8)

This functionality was introduced to allow a user to be able, just by pressing a button on their controllers, to rotate a knob in a flightsim's cockpit.

Let us consider an example, whereby I want to rotate, using my mouse and its left button, a communications dial on my cockpit whose centre is located on the screen at 300, 400.



The statements I need for this are:

```
USE SCREEN_RESOLUTION (1024,768 )
BTN S2 MOUSEROTATE (DL, 300, 400, 40, 30, [MOUSE_LB], CW, 90, 2)
```

So let's go through this in stages. When Button S2 is pressed:

1. DL - The mouse moves to the lower left position of the screen very quickly to get its reference point. Remember the mouse can be anywhere on the screen so we first need to move it to a corner or centre position to reference its movement from there.
2. 300,400 - The mouse uses the centre of the communications dial on the screen to calculate where it is to rotate around.
3. 40 - the radius of the rotation arc it is to move around.
4. 30 - the start angle from the vertical (0 degrees), combined with the previous information determines where macro1 will be activated.
5. [MOUSE_LB] - the mouse left button is held down.
6. CW - the mouse is to be rotated clockwise around the dial
7. 90 - until it reaches 90 degrees from the vertical (0 degrees)
8. 2 - the movement will follow the path of an octagon. Use the smallest number here to get the communications dial to rotate as you want it to.

9. The statement is over so the macro is terminated – i.e. the left mouse button is released.

It's complex I know, but then describing the path of rotation of the mouse is complex! :-)

NOTES ON THE MACRO

1. *The macro will always be held - it's as though there's a* /H *(hold) slash modifier in front of it. The code for the macro will automatically break once the mouse movement is complete.*

2. *Here is the list of allowable key presses....*

- *Simple characters* (a, b, 1, 2, `, *etc.*)
- SHF, ALT, CTL *characters* (A, ALT b, *etc.*)
- *DirectX and POV key presses* (DX1, POVL, *etc.*)
- MOUSE_LB / RB / MB *key presses*
- XFlag *key presses* (X1, X2, etc.)
- USB *key presses*

3. *You cannot use slash modifiers,* DLY( ) *or* RPT( ) *in the Macro.*

NOTES

1. *You cannot group* MOUSEROTATE *statements with other key presses. So:*

BTN S2 a b DLY(30) MOUSEROTATE (*blah blah)* PRNTSCRN

is fine but:

BTN S2 a b DLY(30) **{**MOUSEROTATE (*blah blah*) PRNTSCRN**}**

will generate a compiler error.

2. *You must have a* USE SCREEN_RESOLUTION ( ) *configuration statement present to use this statement, or a compiler error will occur.*

3. *All mouse statements are subject to* RATE *value speed restrictions. If the mouse movement is too slow then reduce the* RATE *statement value. Note that by default if you don't have a* USE RATE (*nnnn*) *statement in your file, it will be set to zero - the fastest response anyway.*

4. *You can for the Macro use macro definitions instead of programming directly into the statement. The most common macros will be* MOUSE_LB *or* MOUSE_RB (*the left and right mouse buttons respectively.*)

5. *Chorded keys will not work in the* MOUSEROTATE *statement. Since everything in the Macro for the mouse rotate is grouped anyway, you should be using* LSHF *and not* SHF.

6. *The Radius can also be expressed in terms of pixels or as a % of the Screen Resolution X axis. This may seem odd at first so let me explain. The radius is a fixed length, normally in pixels. Let's say that we have a screen resolution of 800 by 600, and that we have a radius of 600 pixels (the centre of rotation lying along the bottom of the X axis.) Unlikely, but it could happen. So the radius in % terms is (600/800*100) = 75%. If you now want to use a different resolution, say 1024 by 768, the radius will be 75% of 1024, ie. 768 pixels long. As it's possible to have a radius longer than the screen Y resolution, then it makes sense to express it when using percentages as a percentage of the X axis. There's an important point to make here though. The ratio of the X to the Y axis is critical here. Now for resolutions of 800 by 600, 1024 by 768, 1152 by 864, 1600 by 1200 etc. that ratio is constant and equals 1.333. But if you use a resolution such as 1280 by 1024 (ratio = 1.25) then because the ratio is different, the radius will not scale up exactly. So just be aware of this if you're setting up* MOUSEROTATE *statements using % values rather than pixel values.*

7. *This is a complex statement, and so the chances of creating a compiler error if you forget one of the parts, or a misplaced comma, are pretty high! I implore you then to use the Advanced Mouse Programming Wizard in Foxy to create these statements. For one thing it was a real tricky bit of coding and I'd appreciate it if I knew that it hasn't been in vain!*

8. Logical Programming

8.1 LOGICAL PROGRAMMING - THE BASICS

8.1.1 Understanding flags

Logical programming is all based around the concept of **flags**. So what's a flag? Let's start off with some facts about flags that should totally confuse you, and then we'll make it suddenly very clear with a simple example. Here are the facts then:

- A flag can be on or off.
- There are 32 flags, and they're called X1 to X32.
- A flag doesn't do anything. It's just either on or off.

Confused? (*I sure as hell was when I first started reading up about them!*) So let's use your keyboard's Caps Lock button to explain the concept of flags.

Let's say that your Caps Lock button is called X1. When you press the Caps Lock button, the Caps Lock light on your keyboard switches on, so X1 is on. The keyboard knowing that X1 is on sends out a "W" character instead of a "w" character when you press the "w" key. If I turn off the Caps Lock button by pressing it, the Caps Lock light goes off, so X1 is now off, and now a "w" character is sent to the computer. So we could say that the Caps Lock button doesn't actually *do* anything - it just switches X1 on or off, but it is the fact that X1 is on or off that affects what character the keyboard sends to the computer.

And this is the important concept to understand. A flag doesn't do anything. It's just that it gets turned on or off. You can't see that it is on or off. But you can use the fact that it is on or off to change the way your stick behaves. So what we do in logical programming is to program what happens depending on whether the flag is turned on or off.

8.2 DEFINING LOGICAL FLAGS AND THEIR BUTTON STATEMENTS

Let's start off then by understanding how to turn on or off a flag. Here's the syntax:

Configuration statement:

```
DEF X1 S2
```

Logical programming statement:

BTN X1 /H Fire_rockets

In the above example, we've DEFined a flag called X1 using the DEF statement, and we've stated that it is controlled by button S2. So when button S2 on the joystick is pressed, flag X1 is on, and when button S2 is released, then flag X1 is off.

Once you have a logical flag defined, you can program it with a button (BTN) statement. And in the above example, when button S2 is pressed, your rockets will continually fire. They fire, because you've told them to fire when flag X1 is on, and X1 stays on while button S2 is pressed. It's important to note that logical button statements are subject to the same rules as ordinary button statements with the exception that you can't use /T toggle slash modifiers with them. So they will behave as non-repeating statements, even if the flag is on, unless you use something like a /H hold modifier to hold the logical button down.

Ok, you're probably scratching your head at this stage and asking the question: "*Well why didn't you just have:*

BTN S2 /H Fire_rockets

in your joystick file? Why the need for logical flags?" Well obviously in this example, you're absolutely right, there's no need for logical flags and both statements are equivalent. But we'll come onto an example that you can't achieve with ordinary programming soon - I'm just introducing the syntax for the moment.

It is also possible to define logical flags directly onto digital Type statements and directly with button statements:

RNG 2 5 X1 X2 X3 X4 X5
BTN H1L X8

are perfectly valid statements.

Now there's a slight difference between defining a logical flag using a configuration statement, as opposed to defining it directly onto a button. When you use something like this:

DEF X20 S1

then the flag X20 is on and stays on whilst button S1 is pressed. Now if we have:

BTN S1 X20

in our joystick file (*without a DEF statement*) then it is going to behave as a normal button statement, ie. X20 will turn on and then off even if you keep button S1 pressed. So

```
BTN S1 /H X20
```

will behave the same way as a DEF X20 S1 statement. Note that you cannot have a DEF statement that defines a logical flag, and then have that logical flag on a button or axis statement.

We've said that placing and hence defining a logical flag directly on a button statement is subject to the same programmability as a normal button statement. Therefore you can generate flags using a statement like:

```
BTN S1 KD(X8) DLY(2000) KD(X6) KU(X6 X8)
```

This might not seem like a particular advantage, but if you would like a setup stage, followed by an auto-repeating stage, you could set it up like this:

```
BTN X1 /A Fire_Main_Guns
BTN S1 /H Switch_to_Main_Guns X1
```

Now the /H is used, and this will apply to the X1 part of the statement. So the effect of this is that the Switch_to_Main_Guns is done only once, but the Fire_Main_Guns is auto-repeated by the logical flag. Cool, eh?

NOTES

Be careful about defining the same logical flag both with a DEF *statement, and directly on a button statement. For example:*

```
DEF S4 X1
BTN S2 X1
```

This is effectively the same as saying:

```
DEF X1 S4 OR S2
```

so if either S4 or S2 are being pressed, X1 will be on.

8.3 LOGICAL COMPARATORS

The Logical Comparators consist of the following: AND, NOT and OR which can also be used in conjunction with parentheses.

Configuration statements:

```
DEF X1 S2 OR T6
DEF X2 S4 AND S3 AND H1U
DEF X3 S1 AND NOT X1
```

Logical programming statements:

```
BTN X1 Fire_missile
BTN X2 Engines_off Gather_belongings Eject
BTN X3 AutoPilot
```

Let's look at the first pair:

```
DEF X1 S2 OR T6
BTN X1 Fire_missile
```

Flag X1 can be turned on by either button S2 or button T6. And so pressing button S2 or T6 results in a single missile being fired. Ok we could have gotten the same result with:

```
BTN S2 Fire_missile
BTN T6 Fire_missile
```

So let's look at a situation that can't be programmed directly onto buttons:

```
DEF X2 S4 AND S3 AND H1U
BTN X2 Engines_off Gather_belongings Eject
```

In this example, flag X2 only turns on when button S4 and button S3 are pressed as well as Hat 1 being pushed up. Not something you'll do by accident! But when you do, you'll be landing by the seat of your pants, literally. And in the final example:

```
DEF X1 S2 OR T6
DEF X3 S1 AND NOT X1
BTN X3 AutoPilot
```

we have defined X3 as being turned on by S1, but not if button S2 or T6 are pressed. So button S1 will turn on the autopilot, unless S2 or T6 is pressed.

Note that logical flags follow the same non-repeating behaviour as button statements when programmed directly onto buttons statements.
On a final note before we leave the logical comparators, it is possible to use brackets to group logical statements together. DEF statements can be composed of any combination of AND, OR, and NOT statements, left and right parentheses,

and button or flag references, so long they result in a valid logical equation, for example:

```
DEF X1  (S1 AND NOT S2) OR (X5 AND (H1U OR H2U))
```

is perfectly valid.

8.4 THE LOGICAL TOGGLE

In the examples that we've used so far we've set up logical flags so that they were on when a button, or combination of buttons was pressed, and then the flags turned off when those buttons were released. It's also possible to force a logical flag to behave a little more like a light switch - ie. to stay on when turned on even when the button is released, and to be turned off again when the button is pressed again. In other words, we are toggling between an on and off state for the flag. We do this with a "*" placed after the flag or button.

Configuration statement:

```
DEF X1 S4*
```

Logical programming statements:

```
BTN X1 /A Chaff DLY(30) Flare DLY(30)
```

When button S4 is pressed, flag X1 is turned on. It stays on even when S4 is released, because of the toggle * after it. Only when button S4 is pressed again, will the flag now toggle off. The effect in the example above is that when S4 is pressed and released, you'll start throwing out chaff and flares from your aircraft whilst you try and get out of the way of that SAM and sidewinder (*unlucky day I guess!*) and pressing S4 again will stop the continuous chaff and flare dispensing. (That's if you haven't run out by then of course!)

NOTES

1. *You cannot reference a logical toggle directly on a button. For example:*

```
BTN T6 X1*
```

This will generate a compiler error. (This behaviour is different to the original TM F-22 PRO logical syntax). Logical toggles are only allowed in DEF *statements. So this would be fine:*

```
DEF X1 T6*
```

2. *Note that unlike button statements, the* /T *slash modifier is* ***not*** *permitted with logical button programming statements. So:*

BTN X7 /T a /T b

will result in a compiler error. But this is allowed:

BTN S4 /T X1 /T X2 /T X3

8.5 USING THE LOGICAL DELAY AND PULSE FUNCTIONS

8.5.1 The Delay Function

The DELAY function adds a fixed time between the time that the logical equation produces a true state and the time that the flag actually turns ON. Syntax is:

Configuration statement:

DEF Xflag DELAY(Flag_on_delay) *Logical equation*

Consider this example:

```
DEF X1 DELAY(1000) S1 AND S4
BTN X1 Eject
```

This statement defines X1 to turn on 1 second (1000 milliseconds) **after** S1 and S4 are pressed simultaneously. If either S1 or S2 is released, the delay will be stopped. If X1 has not turned ON yet, it will remain OFF, if it has turned ON, then it will turn OFF immediately. The delay will also be reset, so if S1 and S4 are subsequently pressed again. The entire 1 second delay will start over.

And in this example, it's being put to good use in an Ejection sequence.

NOTES

1. *The logical* DELAY *function is totally different to the* DLY( ) *function. The allowable values in the logical* DELAY *statement are 0 to 327670 (i.e. 5.46 minutes - don't even think about asking why!).*

2. DELAY *statements must appear first in a logical statement. So:*

DEF X1 DELAY(1000) S1 AND S4 *is OK, but:*

DEF X1 X2 AND DELAY(1000) S1 AND S4 *will generate an error.*

8.5.2 The Pulse Function

The PULSE function sends a repeated ON-OFF sequence while the logical equation is true. The two numbers in the function define the ON and OFF periods respectively. Here's the syntax:

Configuration statement:

DEF Xflag PULSE(Time_on Time_off) *Logical equation*

So for example:

DEF X1 PULSE(100 1000) H1U
BTN X1 Trim_up_increase

Pressing HAT 1 up will cause X1 to turn ON immediately for a time period of 1/10th of a second, then OFF for a time period of 1 second, then ON for 1/10th second, etc. That operation would run continuously so long as HAT 1 up was pressed. And in this example we'd increase the trim by 1 stage every 1.1 seconds. Notice that it's 1.1 seconds. If I wanted to increase the trim every second, then the statement would need to be changed to:

DEF X1 PULSE(100 900) H1U

Note that a SPACE character must separate the two numeric values. Using a comma or other separator will cause a compiler error.

NOTES

1. PULSE *statements must appear first in a logical statement. So:*

DEF X1 PULSE(100 1000) S1 AND TG1 *is OK, but*
DEF X1 X2 AND PULSE(100 1000) S1 *will generate an error.*

2. *The actual resolution for any* DELAY, RATE, PULSE *time is around 30 milliseconds, and the minimum effective values is also 30 milliseconds, so any value between 1 and 30 produces about a 30 millisecond delay, 31 through 60 produces a 60 millisecond delay, 61 through 90 produces a 90 millisecond delay etc.*

3. *The allowable values in the logical* PULSE *statement are 0 to 82800000 (i.e. 23 hours).*

8.6 LOGICAL PROGRAMMING EXAMPLES

8.6.1 Toggling a Type 4 statement on and off

Using a Type 4 Digital statement, we can produce repeating, or pulsed characters on the RNG knob like this:

```
RNG 4 1000 a ^ b
```

Now the only way to turn off the pulsed characters is to move the RNG knob into its centre position, where the null character (^) is. But let's say that instead, what we'd like to do, is to be able to depress the RNG knob, ie. button T6, to stop and start the pulsed characters. With logical programming, this is easy and achieved with these statements:

```
DEF X3 T6*
DEF X4 X1 AND NOT X3
DEF X5 X2 AND NOT X3
RNG 4 1000 X1 ^ X2
BTN X4 a
BTN X5 b
```

The way this works is like this. The Range knob, instead of directly producing pulsed characters, now turns on and off the logical flags X1 and X2. Button X4 and 5 generate the a and b characters, but only if the logical flag X3 isn't on, and as buttons T6 toggles X3 on and off, we now have a way of turning on and off the characters generated on the range knob.

8.6.2 A slow trim function

I'd better give credit where credit is due, so this one's courtesy of Mark! We're going to consider that in a flight sim, we use KP7 and KP1 to trim the aircraft's attitude up and down. What we're going to set up, on a hat, is a slow trim function, so that if we press Hat1 Up and release it, then every 5 seconds, a KP7 is generated adjusting the trim, and when we press the hat up again, we stop adding trim. And similarly for Hat1 Down with KP1. And here's how:

```
DEF X1 H1U* AND (NOT X2)
DEF X2 H1D* AND (NOT X1)
BTN X1 /A KP7 DLY(5000)
BTN X2 /A KP1 DLY(5000)
```

Let me explain what's going on here. When Hat 1 is pressed up, it turns on X1 which stays on because we've toggled it with the toggle (*) flag. If X1 is on then we execute the button X1 statement, and we get our KP7 repeating every 5 seconds. When Hat 1 is pressed again, it turns off X1 because its toggled, and we stop generating KP7. It's exactly the same for X2. The inclusion of (NOT X1) ensures that if Hat 1 is pressed down and is generating KP1, and then you press Hat 1 up, then you don't generate KP1 and KP7 at the same time.

There are some files written by Mark Mooney in your Files folder with some more examples of Logical programming.

9. Troubleshooting

9.1 RESETTING THE CONTROLLERS

There are several ways of dealing with problems with the controllers, described hereafter.

9.1.1 In a game: EMPTY_BUFFERS and STICK_OFF

It is (*with great difficulty!*) possible that you *may* produce too many characters from the Cougar, either by pressing too many buttons too quickly (*unlikely*), or by using a file that is poorly programmed (*now this is possible*), causing the joystick to operate erratically. The way that this will manifest itself is that it will appear that the controllers have stalled - the analogue axes will continue to function, but no buttons will appear to work. In this situation, it is possible to clear the buffer memory, without clearing the program within the controllers, so that you can continue to play the game.

Command syntax:

EMPTY_BUFFERS

Example:

```
BTN S2 EMPTY_BUFFERS
```

Obviously this is something that will very rarely be used, and so it would make sense to use this with some logical programming requiring a specific combination of keys to be pressed for some time, before processing it, so that it couldn't be processed accidentally.

```
DEF X1  DELAY(2000) S1 AND S4
BTN X1 EMPTY_BUFFERS
```

In the above example, you'd have to hold down S1 and S4 together, for 2 seconds, before the logical flag X1 is turned on, which would then send a single EMPTY_BUFFERS command to the controllers.

In a similar way you can actually turn off the controllers from within a game, effectively putting the buttons mode into Windows mode, using the STICK_OFF statement.

Command syntax:

```
STICK_OFF
```

Example:

```
BTN S2 STICK_OFF
```

Obviously this is something that will very rarely be used, and so it would make sense to use this with some logical programming requiring a specific combination of keys to be pressed for some time, before processing it, so that it couldn't be processed accidentally.

```
DEF X1  DELAY(2000) S1 AND S4
BTN X1 STICK_OFF
```

Once you use this command, there's no way to turn the sticks back on again, so it's only to be used in an emergency in your file if for some reason your controllers are generating characters and you can't stop it. You will then need to exit your game, and sort out whatever was causing the problem from within Windows.

9.1.2 Within Windows

If you look at Foxy's Cougar menu, you will see menu items allowing you to reset different aspects of the controllers. Here in order of severity then are the steps you should go through as you try to recover your state of mind!

1. Empty buffers

In exactly the same way as before, you can empty the buffer memory, retaining the program within the memory as well as maintaining the controllers in their present mode.

2. Disabling programmed mode

It is very easy from within Foxy to place the controllers into Windows mode, so that they stop generating characters, if for any reason they're spewing out a load of characters. If you're unlucky and that the characters are "F1" function keys, then enjoy the selection of help windows that pop up whilst you reach for the USB plug to unplug the controllers!

3. Clear memory

Windows mode is entered and the memory is cleared of any program.

4. Flash memory

In the case where there is a serious problem with the controllers, so that they are recognised by Windows, but none of the axes or buttons work, or if a new version of the firmware is released, then you can always re-flash or flash upgrade the firmware memory.

5. Call technical support!

If the controllers aren't recognised at all, then either the native Windows drivers aren't installed/functioning correctly, or there is a serious hardware problem. You will then need to contact technical support and determine if the controllers need to be returned.

10. Appendices

APPENDIX 1.
SUMMARY OF THRUSTMASTER STATEMENTS

Button statements and statement modifiers

Statement	Acronym	Description
BTN	Button	Defines the button to be programmed. These are Hat 1 to 4, S1 to S4, T1 to T10, TG1 and 2.
REM	Remark	Any text after a REM statement on a line is ignored by the compiler. Used for comments, titles etc.
RESET_TOGGLES	Toggle reset	Resets a toggle series to the first /T
REVERSE_TOGGLES	Reverse toggles	Produces toggling in the reverse direction
DLY	Delay	Adds a delay between characters or macros.
RPT	Repeat	Repeats a character or macro
( )	Parenthesis grouping	Groups characters/macros together for various statements including digital statement grouping
{ }	Curly Brackets grouping	Groups characters together forcing all their make codes to be generated before their break codes. Similar to holding down a group of keys.
< >	Angle brackets	Forces the statements within them to execute to completion before any other statements are executed
DX	DirectX buttons	DirectX buttons that can be programmed via any button statement
KD KU	KeyDown KeyUp	Allows for greater control of a key press down and up events.
USB	USB keyboard scan codes	Used to generate any make and break character codes for any key.
REVERSE_UD REVERSE_LR REVERSE_DIR	Reverse controller direction	Reverses the appropriate controller's direction e.g. in special HAT statements

Statement	Acronym	Description
NOHOLD, KP5	Affects USE HAT AS statements	NOHOLD stops the Hat statement producing held characters. KP5 adds the KP5 centre position to USE HAT AS KEYPAD statement.
FORCED_CORNERS	Hat corners	Forces hat corner positions to be generated from standard 4 hat positions
S3_LOCK, S3_UNLOCK	Lock S3, Unlock S3	Latches S3 Un-latches S3
SHIFTBTN	S3 assign	Assigns a different button for S3
POV*n*, (n = D, L,R,UL, DL,UR, DR)	Point Of View hat positions	Allows programmatical control of POV hat positions
MOUSE_LB, MOUSE_RB, MOUSE_MB	Mouse buttons	Allows programmatic control of mouse buttons

Slash modifiers and Statement modifiers

Slash modifier	Acronym	Description
/U, /M, /D	Up, Middle Down	Trebles the programmable positions for any hat/button (apart from T7 and T8) by using the throttle's dogfight switch position.
/I, /O	In, Out	Doubles the programmable positions for any hat/button when button S3 on the joystick is pressed (*can't use on S3 itself*)
/P, /R	Press, Release	Separates out for any position programmability for when the controller is pressed, and then when it is released.
/T	Toggle	Toggles forwards through different characters/macros per button press
/H	Hold	Produces held down characters for the duration of the button pressed, irrespective of whether other buttons are pressed. Can be used with other slash modifiers.
/A	Auto-Repeat	Repeats everything on that line.

Configuration statements

Syntax	Description
USE MDEF	Identifies which macro file contains the macro definitions for the current joystick file.
USE *Btn* AS DXn	Assigns DirectX buttons. Replaces PORTB1 IS syntax
USE ALL_DIRECTX_BUTTONS	Assigns Hat1 as POV and all other buttons as DirectX buttons
USE HAT*n* FORCED_CORNERS	Converts an 8 way hat to a 4 way hat, so that the corner positions execute the statements on the adjacent hat positions
USE HAT AS MOUSE, POV, ARROWKEYS, KEYPAD	Define how to use a HAT if not for ordinary BTN statement programming
USE RATE	Default rate at which characters are generated / repeated
USE S3_LOCK USE S3_UNLOCK	Changes S3 to act more like a latched switch
USE S3 AS SHIFTBTN	Define a different button to use for S3 /I, /O
USE HATx_SENSITIVITY	Reduces sensitivity to help in location of corner positions
USE T1_SENSITIVITY	Set the sensitivity of T1
USE FOXY	Used internally by Foxy for various functions, e.g. USE FOXY GRAPHIC USE FOXY README
USE NULLCHR	Defines which character to use as the null character - default is ^
USE KEYBOARD	AZERTY - for Azerty keyboard layout and French game key re-mapping problems
USE PROFILE	Uses profiles saved from the TM Cougar Control Panel
USE CURVE	Define an axis response curve
DISABLE AXIS	Disable an axis
USE SWAP	Swap axes around
USE REVERSE	Reverses an axis
USE AXES_CONFIG	Defines which axes to use, and other attributes of them
USE MTYPE	Assigns mouse control to the microstick, and defines which buttons to use for the mouse buttons
USE MICROSTICK AS MOUSE - NO_BUTTON	Assign the mouse to the microstick, and the left mouse button to T1 (unless - NO_BUTTON is present)
USE Axis_Identifier AS Mouse_Axis	Assigns an axis to control the mouse
USE ZERO_MOUSE	Prevents stuck custom mouse with /I, /O
DISABLE MOUSE	Disable the mouse from being assigned by default
USE SCREEN_RESOLUTION	Used in complex mouse movements

Axes programming

Axis statement	Acronym	Description
JOYX, JOYY	Joystick X and Y	Joystick axes
THR	Throttle	Throttle axis
RDDR	Rudder	Rudder axis
ANT	Antenna knob	Antenna knob axis
RNG	Range knob	Range knob axis
MIX, MIY	Microstick X and Y	Microstick axes
LBRK and RBRK	Left and Right Toe Brakes	Rudder Toe brake axes
MSX, MSY, MSZ	Mouse X, Y, Z	Mouse axes - Z is the mouse wheel
Digital Type statements 1 to 6		Produces keyboard characters from the axes. Also used with mouse, curve and logical flag statements.
FORCE_MACROS	Force macros	Forces the characters/macros in Type 1, 2, 5, 6 statements to always be generated
CURVE	Axis curve	Axis curves - changes axis response and sensitivity
TRIM TO_CURRENT HOLDTRIM	Axis trim	Trims an axis to a specified value
LOCK, UNLOCK, LASTVALUE	Axis lock	Hold an axis value so that it can be used digitally only
SWAP	Swap axes	Swap axes around
REVERSE FORWARD	Axis direction	Reverse an axis, restore normal direction

Advanced mouse statements

Mouse statement	Description
MOUSEXY	Positions the mouse cursor at a particular screen coordinate
MOUSEMOVE	Moves the mouse, relative to its current position
MOUSEROTATE	Programs the mouse to move in rotational movements, allowing cockpit rotaries to be controlled programmatically

Logical statements

Logical syntax	Acronym	Description
DEF	Define	Defines logical flags through equations
BTN	Button	Used to assign virtual button statements to the logical buttons X1 to X32
X1 to X32	Logical flags	Logical flags that are either on or off.
AND, OR, NOT	Logical comparators	Used to construct logical equations
*	Toggle	Toggles a logical flag between its on and off states
DELAY	Delay	Executes a delay after which a logical condition is true
PULSE	Pulse	Produces characters/flag states every *nn* milliseconds

Hardware statements

Syntax	Description
EMPTY_BUFFERS	Empties the buffer memory of the controllers
STICK_OFF	Turns the controllers off in a game

APPENDIX 2. THRUSTMASTER DEFAULT KEY SYNTAX

ESC	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	
`	1	2	3	4	5	6	7	8	9	0	-	=	BSP
TAB	q	w	e	r	t	y	u	i	o	p	[	]	\
CAPS	a	s	d	f	g	h	j	k	l	;	’	ENT	
LSHF	z	x	c	v	b	n	m	,	.	/	RSHF		
LCTL	LALT	SPC								RALT	RCTL		

PRNTSCRN	SCRLCK	BRK

INS	HOME	PGUP
DEL	END	PGDN

	UARROW	
LARROW	DARROW	RARROW

NUML	KP/	KP*	KP-
KP7	KP8	KP9	KP+
KP4	KP5	KP6	
KP1	KP2	KP3	KPENT
KP0		KP.	

NOTES

1. *The Syntax for chorded keys (where you hold down the Shift, ALT or CTRL keys) is* SHF a, ALT b, CTL c. *It is not* LSHF a, LALT b LCTL c

2. *Some keys are reserved:* ( ) { } < > *and need to be programmed with* SHF *statements:*

(= SHF 9
) = SHF 0
{ = SHF [
} = SHF]
< = SHF ,
> = SHF .

APPENDIX 3. USB KEYDOWN AND KEYUP CODES

Example:

BTN S2 /P USB (D04) Rem “a” keydown
/R USB (U04) Rem “a” keyup

Key Name	USB HID code
a A	04
b B	05
c C	06
d D	07
e E	08
f F	09
g G	0A
h H	0B
i I	0C
j J	0D
k K	0E
l L	0F
m M	10
n N	11
o O	12
p P	13
q Q	14
r R	15
s S	16
t T	17
u U	18
v V	19
w W	1A
x X	1B
y Y	1C
z Z	1D
1 !	1E
2 @	1F
3 #	20
4 $	21
5 %	22
6 ^	23
7 &	24
8 *	25
9 (	26
0)	27
Return	28

Escape	29
Backspace	2A
Tab	2B
Space	2C
- _	2D
= +	2E
[{	2F
] }	30
\ |	31
Europe 1 (*See notes*)	32
; :	33
' "	34
` ~	35
, <	36
. >	37
/ ?	38
Caps Lock	39
F1	3A
F2	3B
F3	3C
F4	3D
F5	3E
F6	3F
F7	40
F8	41
F9	42
F10	43
F11	44
F12	45
Print Screen	46
Scroll Lock	47
Break (Ctrl-Pause)	48
Pause	48
Insert	49
Home	4A
Page Up	4B
Delete	4C
End	4D
Page Down	4E
Right Arrow	4F
Left Arrow	50
Down Arrow	51
Up Arrow	52
Num Lock	53
Keypad /	54
Keypad *	55
Keypad -	56

Keypad +	57
Keypad Enter	58
Keypad 1 End	59
Keypad 2 Down	5A
Keypad 3 PageDn	5B
Keypad 4 Left	5C
Keypad 5	5D
Keypad 6 Right	5E
Keypad 7 Home	5F
Keypad 8 Up	60
Keypad 9 PageUp	61
Keypad 0 Insert	62
Keypad . Delete	63
Europe 2 (*See notes*)	64
Keypad =	67
F13	68
F14	69
F15	6A
F16	6B
F17	6C
F18	6D
F19	6E
F20	6F
F21	70
F22	71
F23	72
F24	73
Keyboard Execute	74
Keyboard Help	75
Keyboard Menu	76
Keyboard Select	77
Keyboard Stop	78
Keyboard Again	79
Keyboard Undo	7A
Keyboard Cut	7B
Keyboard Copy	7C
Keyboard Paste	7D
Keyboard Find	7E
Keyboard Mute	7F
Keyboard Volume Up	80
Keyboard Volume Dn	81
Keyboard Locking Caps Lock	82
Keyboard Locking Num Lock	83
Keyboard Locking Scroll Lock	84
Keypad , (Brazilian Keypad .)	85
Keyboard Equal Sign	86

Keyboard Int'l 1 (Ro)	87
Keyboard Intl'2 (Katakana/Hiragana)	88
Keyboard Int'l 2 ¥ (Yen)	89
Keyboard Int'l 4 (Henkan)	8A
Keyboard Int'l 5 (Muhenkan)	8B
Keyboard Int'l 6 (PC9800 Keypad ,)	8C
Keyboard Int'l 7	8D
Keyboard Int'l 8	8E
Keyboard Int'l 9	8F
Keyboard Lang 1 (Hanguel/English)	90
Keyboard Lang 2 (Hanja)	91
Keyboard Lang 3 (Katakana)	92
Keyboard Lang 4 (Hiragana)	93
Keyboard Lang 5 (Zenkaku/Hankaku)	94
Keyboard Lang 6	95
Keyboard Lang 7	96
Keyboard Lang 8	97
Keyboard Lang 9	98
Keyboard Alternate Erase	99
Keyboard SysReq/Attention	9A
Keyboard Cancel	9B
Keyboard Clear	9C
Keyboard Prior	9D
Keyboard Return	9E
Keyboard Separator	9F
Keyboard Out	A0
Keyboard Oper	A1
Keyboard Clear/Again	A2
Keyboard CrSel/Props	A3
Keyboard ExSel	A4
Left Control	E0
Left Shift	E1
Left Alt	E2
Left GUI	E3
Right Control	E4
Right Shift	E5
Right Alt	E6
Right GUI	E7

NOTES

These keys have various legends depending upon the locale for which the keyboard is manufactured. Europe 1 is typically in AT-101 Key Position 42 next to the Enter key. Europe 2 is typically in AT-101 Key Position 45, between the Left Shift and Z keys.

APPENDIX 4. DIFFERENCES BETWEEN ORIGINAL TM FILES AND COUGAR FILES

There are some subtle (and some not so subtle) differences between original TM files supporting the Digital chips, F22, FLCS, FCS, TQS, WCS MkII. This appendix summarises those differences – see the Thrustmaster documentation for more detailed explanations.

1. Changes in key syntax

Old syntax	New syntax
LSFT	LSHF
RSFT	RSHF
none	PRNTSCRN
AUXUAROW, UAROW	UARROW
AUXDAROW, DAROW	DARROW
AUXLAROW, LAROW	LARROW
AUXRAROW, RAROW	RARROW
AUXENT	KPENT
AUX/	KP/
AUXINS	INS
AUXHOME	HOME
AUXPGUP	PGUP
AUXPGDN	PGDN
AUXDEL	DEL
AUXEND	END

2. Slash modifier changes

Slash modifier	Comments
/U	As before
/M	
/D	
/I	As before but /I statements must always appear before /O statements, and /I, /O must be on different lines
/O	
/P	As before
/R	
/T	As before
/A	Now defines auto-repeating statements
/H	As before but characters are generated repeatedly, and in a complex statement, applies to the last statement character
/F	No longer supported
/Q	No longer supported
/N	No longer supported – all statements behave as though they are non repeating unless /H, /A modifiers present

3. Statements no longer supported

Statement	Comments
RAW ( )	Replaced by USB ( ) *(HID codes)* but easier to use KD( ) and KU( ) statements
BTN MT	Use Type 5 statements instead – they're more powerful
BTN T11 – T14 (they no longer exist)	The microstick is like a 2 axis joystick – it isn't a 4 button controller. So can be programmed digitally with Type 1 to 6 digital statements
USE RCS	No longer needed
USE TQS	No longer needed
USE WCS	No longer needed
USE RCSPRO	No longer needed
USE NOMOUSE	No longer needed
USE NOTHR	No longer needed
USE MTYPE (B or C)	Mouse types no longer supported, although 3 button mice statements can be generated.

4. File extensions, file names

The joystick file must end in ".tmj" and the macro file in ".tmm". Also, joystick file names as well as macro file names can contain spaces and are not restricted to 8 character name lengths. But as before the joystick file must be in the same folder/directory as the macro file. By default this is Foxy's Files folder.

5. Default actions

The compiler will setup some default actions for you with your files, dependant on how you've setup your preferences in Foxy. These options are overridden either if deselected in Foxy, or if you've programmed your files to behave differently.
They are:

- Hat 1 or a Hat of your choice as a Point Of View (POV) Hat
- TG1 as DX1, S2 as DX2 – i.e. as DirectX buttons, so these buttons will have their functions assigned within a game if the game supports it.
- If no USE MDEF line is present, and the joystick file contains macros, then the compiler will look for its macros from a file having the same name as the joystick file, but with the extension ".tmm"
- The Microstick will control mouse movement, with T1 on the microstick operating as a left hand mouse button.

The reasons for these defaults are to make it easier for beginners to get into programming. So if a user has a joystick file that just has say:

```
BTN S1 Autopilot
```

and a macro file with just:

```
Autopilot = a
```

then they will be able to download the joystick file straight away and still find that their trigger works, that the mouse works, etc.

The only danger here is that if the files are passed onto someone else, who has changed their default settings in Foxy, say to produce an error if no USE MDEF line is found, then the file won't work for that user.

6. Digital vs. Analogue axes

With the original TM files, an axis could be programmed either digitally, thus producing keyboard characters, or left as the default analogue mode, whereby its function was assigned by the game (eg. Throttle axis = thrust in a flight simulator). With the HOTAS Cougar, axes can be both analogue and digital at the same time if required. To have a pure digital axis, its analogue function must be disabled with the DISABLE statement. Also note that with the Cougar, you do not need to change anything in the Control Panel's Gaming Options applet, if you want to use an axis as purely digital, which you had to do with the original TM controllers.

7. Type 1 digital statements

The syntax for these has changed – see the section in this reference book detailing this.

8. Throttle not present

If you have a file written for a joystick and throttle, and the throttle isn't present, then with /U, /M, /D statements, only the /M statement will be compiled – the /U, /D will be ignored. Throttle axes statements will also be ignored.

9. Macros - disallowed characters

You **cannot** use the following characters in macro names:

= < > { } ( ) ^ , *spaces*

10. RPT

If you have a macro like this: Macro1 = a b c

and a statement like this: BTN S2 RPT (3) Macro1

then when S2 is pressed you will get: a a a b c

To avoid this, either enclose the macro or the characters in its definition with parentheses brackets, i.e.:

BTN S2 RPT (3) (Macro1)

or

BTN S2 RPT (3) Macro1

Where: Macro1 = (a b c)

11. The // comment characters

With the digital chips, you could use // characters instead of REM statements. These are not supported in Foxy.