Version 6

31 August 2007

For latest manual updates use The Application Support Centre.

Programmer User Manual

Table of contents

1. Getting Started

1.1 Introduction

The aim of this document is to provide the reader with information regarding all functions and illustrate how to best use the different aspects of the API.

The full function listing for the API can be seen in the programmer.h file.

[TOP]

1.2 Backwards Compatibility

The Programmer API changes significantly from version to version, however every effort is made to ensure that your existing plugins will migrate to the next version format with as little effort as possible.

Note: In the initial V6.0 release all graphical functions (GUI / draw / etc) were disabled.

[TOP]

1.3 Naming Conventions

The Programmer interface uses a standardised naming convention for all functions. The naming convention can be broadly split into two sections, the action identifier and the object identifier.

The action identifier is a 3-letter code that comprises the first three characters of a function name. The action identifier is intended to illustrate the core purpose of the function. The action identifiers are:

- **QPO**: Override Standard Code - define a function in the plugin that overrides the standard default behaviour inside Paramics. Each of the key parts of the model can be replaced by your own logic;
- **QPX**: Extend Standard Code - define a function in the plugin that adds to the functionality in Paramics. It can be triggered by one of a large number of events i.e. when the network is loaded, saved refreshed, or at the start/end of each timestep etc.
- **QPG**: Get a value from the Standard Code - retrieve data from within either the simulation or graphics engines inside Paramics; and
- **QPS**: Set a value in the Standard Code - set a data value or change or add to the view displayed.

The QP_ prefix is followed by a 3-character code dividing the interface into functional groupings, this is the object identifier:

- **NET** - Network
- **CFG** - Configuration
- **STA** - Statistics
- **DRW** - Drawing

- **UTL** - Utility
- **POS** - Position
- **DMD** - Demand
- **NDE** - Node
- **LNK** - Link
- **CAT** - Link category
- **ZNE** - Zone
- **VHC** - Vehicle
- **TGV** - Tagged vehicle
- **BUS** - Bus
- **CLK** - Clock/timing
- **SCO** - Single-carriageway overtaking
- **GUI** - User-interface controls
- **CFM** - Car following model
- **LCM** - Lane changing model
- **RTM** - Routing model
- **VTP** - Vehicle type
- **DTC** - Detector
- **DTL** - Loop-type detector
- **DTI** - Detector using index
- **BCN** - Beacons
- **BCI** - Beacon using index
- **BSR** - Bus Route
- **BST** - Bus Stop
- **CPK** - Car park
- **RMP** - Ramp
- **SLP** - Slip lane
- **SIG** – Traffic signals

[TOP]

1.4 Data Types

All Paramics data objects are typed so your code can be type checked by the compiler at compilation time reducing errors. This also helps in the debugging process and provides the user with more readable code.

The current data types are:

- NODE

- LINK
- ZONE
- VEHICLE
- BUSSTOP
- DETECTOR
- CARPARK
- BEACON
- LOOP
- SLIP
- RAMP

In addition to the base data structures mentioned above you can now associate your own data structures with Paramics objects. For example imagine you need to store the details of a vehicles route choice with each vehicle in the simulation (like worked example 6).

In V6 you can create your own C data structures with their own variables/attributes. These data structures can then be stored directly with Paramics objects removing the need for lookup tables etc. and simplifying your code.

[TOP]

1.5 Compiler Setup

Paramics Programmer is implemented via “dll” files (Dynamic Linked Libraries). This means that when you have a Programmer license, Paramics Modeller will load user “dll” libraries representing the user plugins.

Provided with the base plugin are Microsoft™ Visual C++ (MSVC) project and workspace files: these are used for compiling on the PC platform. To edit the plugin load the “Plugin.dsw” file in to MSVC, click on the File View tab on the left-hand panel, then expand the “Plugin files” folder and the “Source Files” folder then double click on plugin.c, this will then load the file.

When you have made your modifications to the plugin (by editing plugin.c), you will then need to compile a new version. There are two types of configuration available:

- **Debug compile** will allow you to trace through your API code at run time for the purpose of finding errors in your code
- **Release compile** is optimised for faster execution at run time.

To select the type of compile that MSVC will perform select the Set Active Configuration option from the Build menu. This will bring up a window from which you can select either Debug or Release.

You are now ready to compile Paramics Programmer. This is done by selecting Build plugin.dll from the Build Menu (or using the F7 shortcut key).

If you are set to compile a “Debug” version then the resulting “dll” will be placed in a subdirectory called “Debug”
Similarly if you are compiling a “Release” version then the resulting “dll” will be placed in a subdirectory called “Release”.

[TOP]

1.6 Loading Plugins

Plugins are loaded and unloaded dynamically i.e. each time the software is initialised or a new network is loaded; this means that your plugins can be network specific.

Plugins are loaded by either directly specifying the plugin DLL file, or specifying the path to the file to load. This can be done in a number of ways: to allow the user control over this process a search hierarchy is defined i.e. the order in which Paramics Modeller will try to look for a plugin.

The hierarchy is:

1. Place the DLL file in the default plugins directory

<PARAMICSHOME>\plugins\

i.e.

C:\Program Files\Paramics\plugins\

2. Place the full path to the plugin in the default plugins load file

<PARAMICSHOME>\plugins\

i.e.

C:\Program Files\Paramics\plugins\

3. Place the name of the plugin is a network specific file called ‘programming’. The new 'programming' files can also be application specific e.g.

- programming
- programming Modeller
- programming Processor
- etc

Programming files are really designed to let you specify specific plugins for specific networks.

Note: PARAMICSHOME is defined as an environmental variable. You can open up a command prompt and type **SET** to check this value in windows.

Tip: Using a programming file gives you the option to specifically enable or disable a QPX function from being called improving execution performance and run times for your plugins. Plugins loaded with a programming file must be placed in the root Paramics directory i.e. the directory where the modeller.exe file is located.

[TOP]

1.7 Adding New C Files

You can change the list of C files that a plugin uses by editing the MSVC project files. To change the list of C files you will need to do the following:

- Load the “Plugin.dsw” file into MSVC.
- Select the file view tab in the bottom left of the screen.
- In the file view expand the “Plugin files” folder and select and expand the “Source Files” folder.
- Now to add a C file right click on “Source Files” and select “Add Files to Folder”.
- To delete a C file left click on the file and pressing the delete key.

[TOP]

2. Starting A New Plug In Project

2.1 Quick Start

A good way to start a new project is to take a copy of the “base” plugin directory.

Once you have copied the directory, it is a good idea to give the plugin project a new name. To save confusion it is usually a good idea to give the plugin and the plugin directory the same name.

For example if your new project is called “plugin1” you should perform the following steps:

- Copy the “programmer/plugins/base” directory to “programmer/plugins/plugin1”
- Rename the plugin to “plugin1”

[TOP]

2.2 Renaming your plugin

You can change the name that the “dll” is compiled to by editing the MSVC project file. To do this you will need to load the “Plugin.dsw” file into MSVC. To change the name first select the project settings (Project>>Setting or ALT+F7). This will bring up the project settings window, in this window select the “Link” tab. Then in the “Settings For” select “Win32 Debug”.

You can now enter the name of the “dll” to be generated in the “Output file name” field. So in our example this should be changed to “Debug/plugin1.dll”. You will now need to repeat this for the “Win32 Release” settings so that the “Output file name” is set to “Release/plugin1.dll”.

[TOP]

3. QPO Override Functions

3.1 QPO Overview

Override functions let the user replace algorithms / routines in the core Paramics model with their own code. This is useful for academic research where the researcher can utilise the power of the Paramics system while still maintaining the flexibility to experiment as needed.

[TOP]

3.2 Function Summary

The following text provides a brief summery of each override function purpose and how each can be used by the user to override the existing algorithms in the Paramics core model:

- **void qpo_CFM_behaviour(VEHICLE* vehicle)** – This function lets the user define how a vehicles behavioural characteristic are assigned;
- **float qpo_CFM_constrainedAcceleration(VEHICLE* vehicle, LINK* link, float speed, float dist) -** This function lets the user override the car following mode implemented in Paramics;
- **float qpo_CFM_curveSpeed(float r, LINK* link)** – This function defines how vehicles react to curved road segments and how the radius of the curve effects the vehicle speed;
- **float qpo_CFM_followingAcceleration(VEHICLE* vehicle, VEHICLE* aheadvehicle, float gap, LINK* link)** - This function lets the user override the car following mode implemented in Paramics;
- **float qpo_CFM_gradientAcceleration(int type, int age, float speed, float gradient, float acc)** - This function lets the user override the car following mode implemented in Paramics;
- **float qpo_CFM_headwayFactor(VEHICLE* vehicle, LINK* link)** - This function allows the user to influence a specific vehicles target headway;
- **float qpo_CFM_linkSpeed(LINK* link, VEHICLE* vehicle)** - This function defines the maximum obtainable speed the given vehicle can achieve on the given link;
- **int qpo_CFM_lookAheadCount(int awareness)** - This function defines how far in front a vehicle can see, i.e. 2 cars , 3 cars etc.
- **float qpo_CFM_minimumGap(void)** - This function defines the minimum gap between two vehicles;
- **float qpo_CFM_rampMergeHeadway(VEHICLE* vehicle)** – This function defines how a vehicles headway is effected in and around a ramp merging area;
- **float qpo_CFM_rampRevertDistance(VEHICLE* vehicle)** – This function defines when a vehicles headway returns to normal conditions after passing or merging from a ramp;

- **float qpo_CFM_safeDistance(VEHICLE* vehicle, float speed)** – This function defines how the safe distance between two vehicles is calculated;
- **float qpo_CFM_stoppingDistance(VEHICLE* vehicle)** - This function defines how the stopping distance for a vehicle is calculated;
- **float qpo_CFM_thresholdDistance(VEHICLE* vehicle)** – This function defines the threshold distance for a vehicle;
- **float qpo_LCM_forceMergeDistance(VEHICLE* vehicle, VEHICLE* remvehicle)** – This function defines the distance between two vehicles where the minor vehicle can force into the major traffic stream;
- **Bool qpo_LCM_gapExists(VEHICLE* vehicle, VEHICLE* aheadvehicle, VEHICLE* behindvehicle, LINK* link, Bool u)** – This function defines gap acceptance for a vehicle;
- **void qpo_LCM_laneUsage(VEHICLE* vehicle, LINK* link, int *loLane, int *hiLane)** – This function defines how a vehicle lane range is effected in different types of links;
- **Bool qpo_LCM_moveIn(VEHICLE* vehicle, VEHICLE* infrontvehicle[], VEHICLE* behindvehicle[], LINK* link)** – This function defines the logic for a vehicle to change lane to a lane with a lower index;
- **Bool qpo_LCM_moveOut(VEHICLE* vehicle, VEHICLE* infrontvehicle[], VEHICLE* behindvehicle[], LINK* link)** – This function defines the logic for a vehicle to change lane to a lane with a higher index
- **int qpo_LCM_overtakeTime(VEHICLE* vehicle)** – This function sets the time required for a vehicle to initiate a lane changing manoeuvre;
- **int qpo_LCM_passTime(VEHICLE* vehicle)** – This function sets the time required for a vehicle to complete a lane changing manoeuvre;
- **int qpo_LCM_resetTime(VEHICLE* vehicle)** – This function sets the time required to elapse before a new lane changing manoeuvre can be initiated;
- **float qpo_LCM_signpostAllowDistance(VEHICLE* vehicle, float signposting)** – This function sets the distance before a signpost where vehicles will create gaps to let other driver change lanes;
- **float qpo_LCM_signpostMergeDistance(VEHICLE* vehicle, float signposting, Bool urgent)** – This function sets the (urgent) distance inside a signpost where vehicles will create gaps to let other driver change lanes;
- **int qpo_RTM_decision(LINK* link, VEHICLE* vehicle)** – This function is used to calculate route choice through the network on a link by link basis;
- **Bool qpo_RTM_enable(void)** – This function is used to enable user defined route choice;
- **void qpo_RTM_nextLink(LINK* link, VEHICLE* vehicle, int nextout, LINK* *nextlink, int *newdestp)** – This function is used to set a vehicle stow node look ahead;
- **Bool qpo_SCO_gapExists(VEHICLE* vehicle, VEHICLE* aheadvehicle, VEHICLE* behindvehicle, LINK* link, Bool u)** – This function is used for sing carriageway overtaking, to check whether or not a gap in the oncoming traffic exists;
- **Bool qpo_SCO_pullIn(LINK* link, VEHICLE* vehicle)** – This function defines the logic for a vehicle to change lane back out of the opposite side of the carriageway to avoid an head-on collision with on coming traffic;

- **Bool qpo_SCO_pullOut(LINK* link, VEHICLE* vehicle)** – This function defines the logic for a vehicle to change lane onto the opposite side of the carriageway to overtake a slow moving vehicle;

[TOP]

4. QPX Extending Functions

4.1 QPX Overview

Extending functions let the user add new code to the Paramics simulation; they act as entry points to the Paramics software where the user can “hook” in their own code extending the model. In many cases a programmer plugin will have at least one QPX function. There are many more entry points to the simulation and new functions to integrate with network editing and object selection in the main graphics display.

[TOP]

4.2 Function Summary

The following text provides a brief summery of each extending function indicating where the function is placed in the simulation loop:

- **void qpx_BUS_stopping(VEHICLE *vehicle, BUSSTOP *stop)** - called for each PT vehicle in the network when it stops at a PT stop;
- **void qpx_CFG_parameterFile(char* filename, int count)** – called when an API file is specified by the user;
- **void qpx_CLK_startOfSimLoop(void)** - called at the start of the simulation loop for each iteration or timestep;
- **void qpx_CLK_endOfSimLoop(void)** - called at the start of the simulation loop for each iteration or timestep;
- **void qpx_DRW_instrumentView(void)** – called at the end of each simulation loop. This function lets the user add their own 2D graphics to the Paramics display;
- **void qpx_DRW_modelView(void)** – called at the end of each simulation loop. This function lets the user add their own 3D graphics to the Paramics display;
- **void qpx_GUI_keyPress(int key, int ctrl, int shift, int left, int middle, int right)** – called when a key and / or mouse button is pressed in the main Paramics display;
- **void qpx_GUI_keyRelease(int key, int ctrl, int shift, int left, int middle, int right)** – called when a key and / or mouse button is released in the main Paramics display;
- **void qpx_GUI_layer(char *name, Bool on)** – called when a checkbox in the API Layers panel of a Paramics application Layer Selector is toggled;
- **void qpx_GUI_parameterToggle(char* filename, int index, char* label, Bool value)** – called when a Boolean parmeter value is read from a specified parameters file;
- **void qpx_GUI_parameterValue(char* filename, int index, char* label, Bool value)** - called when a float parmeter value is read from a specified parameters file;

- **void qpx_GUI_tool(char *name, Bool on)** – called when an API tool in the Tools menu of a Paramics application is toggled;
- **void qpx_LNK_timeStep(LINK* link)** - called for each link in the Paramics network at each simulation timestep;
- **void qpx_NET_close(void)** – called once when the Paramics network is closed;
- **void qpx_NET_complete(void)** – called once when the simulation end time is reached;
- **void qpx_NET_feedback(void)** – called at each feedback period after the routes have been recalculated;
- **void qpx_NET_hour(void)** – called once after each full hour of simulation time;
- **void qpx_NET_minute(void)** – called once after each full minute of simulation time;
- **void qpx_NET_period(void)** – called once at each period change during the simulation run;
- **void qpx_NET_postOpen(void)** – called once after the network data has been processed;
- **void qpx_NET_postSave(void)** – called once after the network data is saved;
- **void qpx_NET_preOpen(void)** – called once before the network data has been processed;
- **void qpx_NET_preSave(void)** – called once before the network data is saved;
- **void qpx_NET_refresh(void)** – called once after the network data is refreshed;
- **void qpx_NET_reload(void)** – called once after the network data is re-loaded;
- **void qpx_NET_second(void)** – called once after each full second of simulation time;
- **void qpx_NET_start(void)** – called once when simulation run is started;
- **void qpx_NET_timeStep(void)** – called at the start of each time step during the simulation run;
- **void qpx_NET_timeStepPostLink(void)** – called at the end of each time step during the simulation run;
- **Bool qpx_STA_enable(void)** – called once to force creation of a output statistic directory;
- **void qpx_TGV_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)** – called when each tagged vehicle arrives at it's destination zone;
- **void qpx_TGV_beacon(VEHICLE* vehicle, LINK* link, BEACON* beacon)** – called when each tagged vehicle passes a VMS beacon;
- **void qpx_TGV_detector(VEHICLE* vehicle, LINK* link, DETECTOR* detector)** – called when each tagged vehicle crosses a loop detector;
- **void qpx_TGV_laneChange(VEHICLE* vehicle, LINK* link, int lane1, int lane2)** – called when each tagged vehicle changes lane;
- **void qpx_TGV_move(VEHICLE* vehicle, LINK* link, float distance, float speed)** – called when each tagged vehicle is moved forward on a link in the simulation;
- **void qpx_TGV_targetHeadway(VEHICLE* vehicle, LINK* link)** – called when a tagged vehicle reaches is target headway;
- **void qpx_TGV_targetSpeed(VEHICLE* vehicle, LINK* link)** – called when a tagged vehicle reaches it's target speed;
- **void qpx_TGV_timeStep(VEHICLE* vehicle)** - called every timestep for any vehicle for which the user tag is non-zero. It can be used in conjunction with the other TGV functions to improve the performance of a Plugin.

- **void qpx_TGV_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)** – called when a tagged vehicle transfers from one link to another in the simulation;
- **void qpx_UTL_report(char *message)** – called when a message is emitted from Paramics;
- **void qpx_VHC_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)** – called when a vehicle arrives at it's destination zone;
- **void qpx_VHC_beacon(VEHICLE* vehicle, LINK* link, BEACON* beacon)** – called when a vehicle passes a VMS beacon;
- **void qpx_VHC_detector(VEHICLE* vehicle, LINK* link, DETECTOR* detector)** – called when a vehicle passes a loop detector;
- **void qpx_VHC_laneChange(VEHICLE* vehicle, LINK* link, int lane1, int lane2)** – called when a vehicle changes lane;
- **void qpx_VHC_move(VEHICLE* vehicle, LINK* link, float distance, float speed)** – called when a vehicle moves forward on a link in the simulation;
- **void qpx_VHC_release(VEHICLE* vehicle)** – called when a vehicle is released from a zone;
- **void qpx_VHC_targetHeadway(VEHICLE* vehicle, LINK* link)** – called when a vehicle reaches is target headway;
- **void qpx_VHC_targetSpeed(VEHICLE* vehicle, LINK* link)** – called when a vehicle reaches it's target speed;
- **void qpx_VHC_timeStep(VEHICLE* vehicle)** – called for each vehicle in the simulation each timestep;
- **void qpx_VHC_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)** – called for each vehicle in the network when it moves from one link to another;
- **void qpx_LNK_vehicleTimeStep(LINK* link, VEHICLE* vehicle)** – called for each vehicle on each link in the network each simulation time step;
- **void qpx_ZNE_timeStep(ZONE* zone)** – called for each zone in the network each simulation time step.

[TOP]

5. QPS Set Functions

5.1 QPS Overview

The setting functions let the user set values, states and action for objects in the Paramics model. This section gives a brief overview of the setting functions provided by the API.

[TOP]

5.2 Drawing Functions

The majority of new QPS functions relate to drawing routines and adding 2D/3D graphics to the Paramics graphics engine. Drawing functions come is three basic styles these are:

- **Setting States:** The Paramics OpenGL graphics engine is a state machine, like all other graphics API's. Once a graphics state is set the graphics engine remains in that state until it is changed, this is why it is always good practice to execute graphics state changes in pairs i.e. one call to change the state and a second call to change it back to the original value. Some examples of state calls are:
    - **qps_DRW_colour(int c)** – set the current drawing colour;
    - **qps_DRW_linewidth(float width)** – set the current line width;
    - **qps_DRW_rotate(float angle, char axis)** – set the rotation transformation about the given axis;
    - **qps_DRW_loadDrawingMatrix(void)** – re-set the depth of the matrix transformation stack; and
    - **qps_DRW_vehicleColour(VEHICLE* vehicle, int colour)** – set the colour of a specific vehicle;
- **Pre-defined Library Calls:** Library calls are more complex graphics processes, wrapped up into a single function for ease of use. This typically will use one or more Paramics objects. Some example of library calls are:
    - **qps_DRW_filledCircle(float x, float y, float z, float r)** – draw a filled circle of radius r with it's centre point at x,y,z;
    - **qps_DRW_shadeLink(LINK* link, int colour)** – shade the given link the specified colour (the body of this function has about 300 lines of code);
    - **qps_DRW_vehicleTag(VEHICLE* vehicle, int colour, int type, float size, char *format, ...)** – draw a marker and/or text next to a vehicle.
- **Low Level Calls:** Low-level calls are those which operate on graphics primitives at the base level. These are very close to native OpenGL graphics calls. Some example of low level calls are:
    - **qps_DRW_vertex(float x, float y, float z)** – draw a single point in 3D space, similar to a glVertex3f() call in OpenGl; and
    - **qps_DRW_objectBegin(int object)** – start a new drawing object, similar to a glBegin() call in OpenGl;

You can also add native OpenGL calls into your plugin code providing you include the relevant OpenGL header files and link against the public OpenGL libraries.

[TOP]

5.3 GUI Functions

A range of Graphical User Interface functions are available to let you add and access items to the API Layers panel in the Layer Selector, and the Tools menu in a Paramics application. Custom code can then be executed whenever one of these items is toggled.

- **qps_GUI_addTool(char *name)** – this function lets the user add a button to the Tools menu in a Paramics application; it's corresponding QPX function, qpx_GUI_tool(char *name, Bool on), is called when the menu item is toggled; and

- **void qps_GUI_addViewLayer(char *name)** – This function adds a check box to the API Layers panel in the Layer selector of a Paramics application; it's corresponding QPX function, qpx_GUI_layer(char *name, Bool on), is called when this item is toggled.

If your are using a separate user interface API for your plugin code i.e. Java or MFC you can use these functions to open your dialogs/user interfaces.

[TOP]

5.4 User Data Functions

User data functions are used to associate user defined data structures with objects in the Paramics code e.g. a zone.

[TOP]

6. QPG Get Functions

6.1 QPG Overview

Getting functions let the user query the value and state of objects in the Paramics model.

[TOP]

6.2 Bus/PT Data

Bus/PT data functions have the BSR and BST object codes. The bus/pt interface in Paramics Modeller has been completely reworked to provide more flexibility; this has followed through to the API. Bus/PT data function let you query specific information about the PT services and vehicle present in the network but also let you dynamically control when PT vehicles should be released, which PT line they should follow, and how long they should wait at a PT stop.

[TOP]

6.3 Configuration Data

Configuration data functions have the CFG object code. The configuration functions let you access the elements of the simulation configuration, essentially the network configuration file.

[TOP]

6.4 Demand Data

Demand data functions have the DMD object code. Demand object functions let you query demand information from the paramics model i.e. the number of matrices, the demand periods, individual OD cell values etc. Also you can set OD demand levels using the corresponding QPS functions.

[TOP]

6.5 Drawing Data

QPG drawing functions are mostly related to querying states in the graphics engine i.e. to query/store the current state of an aspect of the graphics engine before and after you change it.

[TOP]

6.6 User Data

User data functions exist for all core object types, as seen above user data can be associated with Paramics objects to help extend the core model.

[TOP]

6.7 Category Data

Category data functions have the CAT object code. Category function can be used to query attribute of the link categories in the paramics model. Categories are read only objects and no QPS function exist for them.

[TOP]

6.8 Positional Data

Positional data functions have the POS object code. Positional functions are used to retrieve 3 dimensional position data about the location and orientation of Paramics objects in the 3D paramics world. Positional data can be retrieved for core model objects like vehicles, nodes, zones, kerb point etc. but also for world positions like, the current viewpoint and cross hair.

[TOP]

6.9 Statistics Data

Statistic data functions have the STA object code. Statistics functions provide a series of general performance statistics together with more detailed link/lane based statistics.

[TOP]

6.10 Utility Functions

Utility functions have the UTL object code. Utility functions are provided to aid plugin development and provide services such as, access to random numbers and distribution algorithms, file access and permission checking, units switching, environment variables, and licensing.

[TOP]

6.11 Ramps & Slip Lanes

Ramp and slip lane access is with the RMP and SLP object codes. User can query the vehicles on ramps and slip lane for use in car following and lane changing models.

[TOP]

7. Function Reference

For a complete function list please refer to the CHM files. If you are reading the manual or viewing the PDF then the CHM files can be either downloaded from the customer area documents page or accessed from a PC with Paramics installed via..

START MENU -> ALL PROGRAMS -> PARAMICS -> MANUALS -> PROGRAMMER -> Programmer V6 Help File

[TOP]