

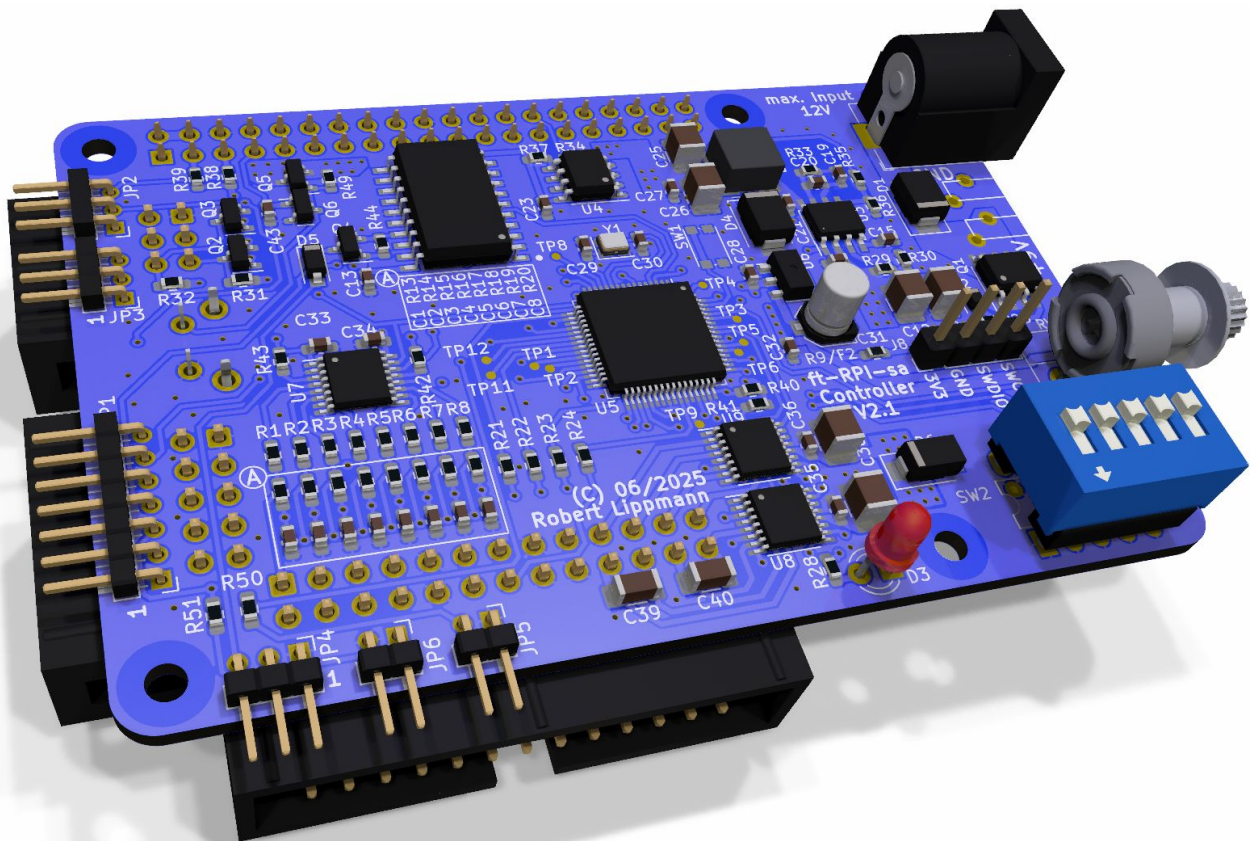
# ft-RPI-sa Controller

(fischertechnik-Raspberry PI-stand-alone Controller)

Anleitung

HW Version 2.x

MMBasic Version 5.05.05



MMBasic © Geoff Graham

Hardware & S32K14x Adaptierung © Robert Lippmann

Dokument V0.2 vom 21. Juni 2025

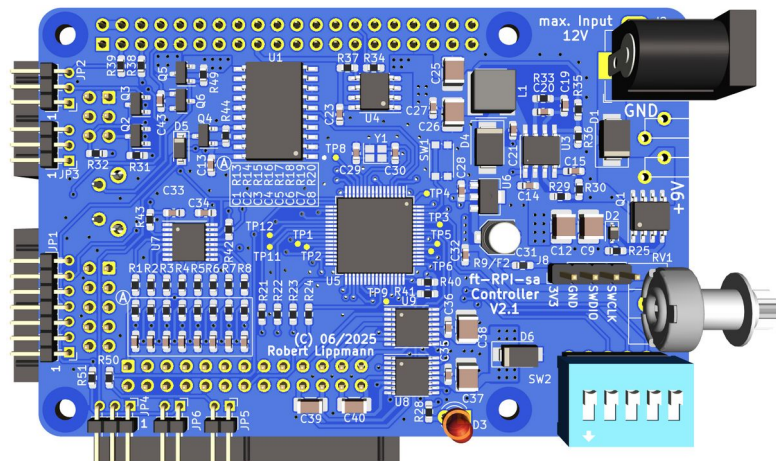
Projektkontakt: [ft.mcu.prj@gmail.com](mailto:ft.mcu.prj@gmail.com)

# Einleitung

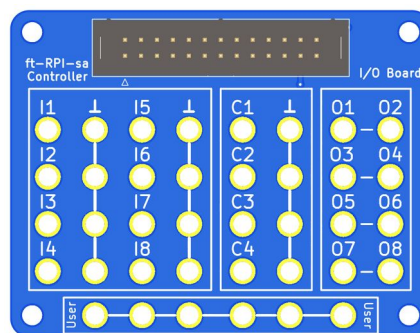
Der ft-RPI-sa Controller ist ein kleiner Computer mit dem **fischertechnik**, Lego oder andere Technikmodelle, die mit 9V versorgt werden, gesteuert werden.

Der Controller besteht aus der Hauptplatine (auch Mainboard genannt), die all die benötigten Bausteine für die Steuerung beinhaltet, sowie einer zusätzlichem Ein-/Ausgabe Platine (I/O-Board) mit allen Anschlüssen für das zu steuernde Modell.

Die Hauptplatine, die optional auch als Raspberry Pi HAT verwendet werden kann, wird entweder durch ein passendes Batterie-Set oder Netzteil mit Spannung versorgt.



Das Mainboard



Das I/O Board

Das Mainboard beinhaltet einen Mikrocontroller (MCU), eine Echtzeituhr (RTC) mit Backup Batterie, Spannungsreglern, Ausgangstreibern (H-Brücken) und viele Anschlüsse sowie Konfigurationsbrücken.

Die benutzte MCU ist ein S32K144 Mikrocontroller der Firma NXP, die einen 32bit ARM CPU Kern des Typs M4F beinhaltet, welcher mit 80MHz betrieben wird. Neben der CPU besitzt die MCU viele weitere Peripheriemodule wie Flash Speicher (Programmspeicher), RAM Speicher (Arbeitsspeicher), UART, I2C, SPI, CAN, ADC, Timer etc.

Statt einer S32K144 MCU ist es auch möglich, einen anderen Typ der S32K1 Familie, im 64pin LQFP Gehäuse, zu verwenden (z.B. S32K118, S32K142(W) oder S32K146). Die Unterschiede der einzelnen MCUs, die die Leistung oder Fähigkeiten, des ft-RPI-sa Controllers beeinflussen, sind nachfolgend dargestellt:

MCU Typ	Flash Speicher	RAM Speicher	Max. CPU Geschwindigkeit	CPU Kern
S32K118 <sup>1</sup>	256kB	25kB	48MHz	Cortex M0+
S32K142(W)		32kB	112MHz(80MHz)	Cortex M4 incl. FPU
S32K144(W)	512kB	64kB		
S32K146	1024kB	128kB	112MHz	

MCU Unterschiede

Wenn Du einen detaillierteren Überblick der K1 Familie haben willst, besuche die [NXP Webseite](#). Dort gibt es auch das "Reference Manual", das jedes einzelne Bit der MCU erklärt.

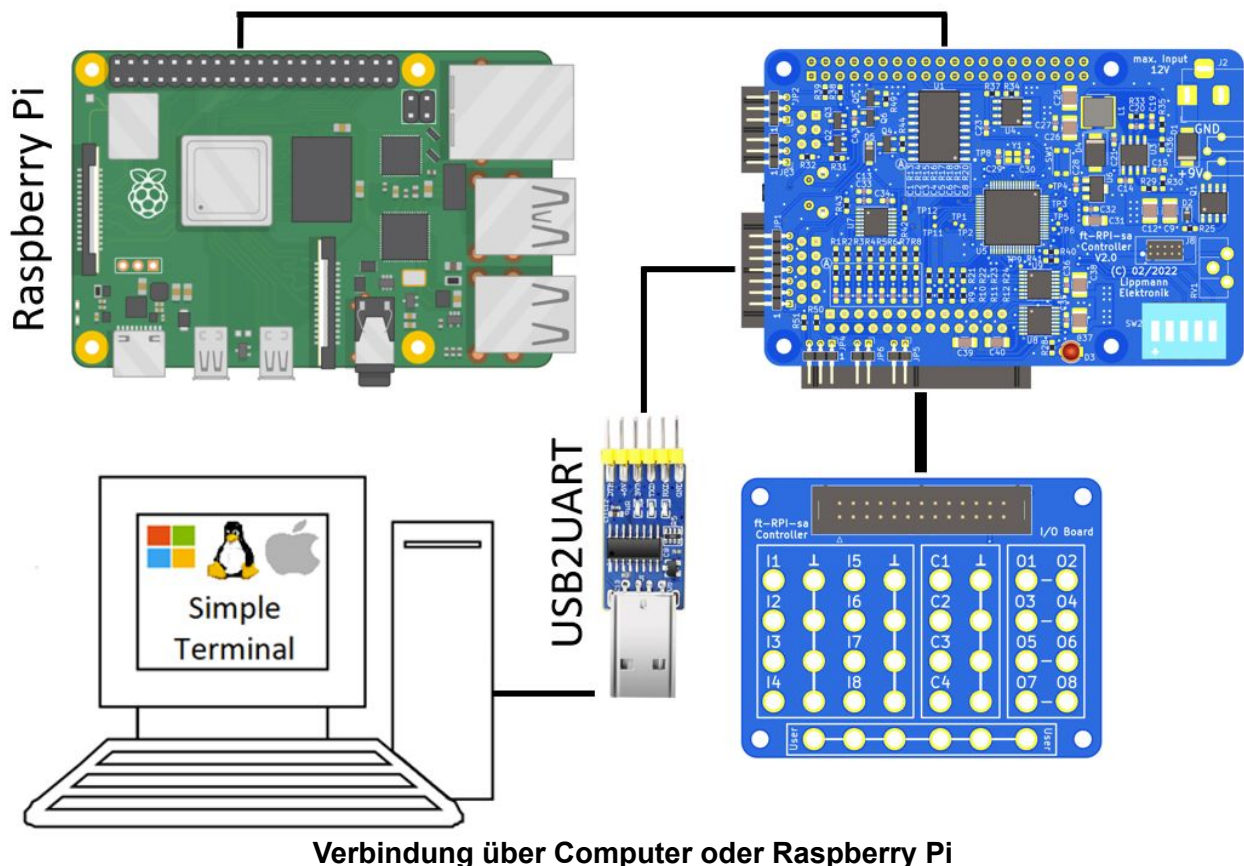
<sup>1</sup> Die MCU S32K118 hat keine FPU und somit sind die Fließkommaoperation deutlich langsamer, da diese in Software abgearbeitet werden müssen, und somit verweise ich auf diese MCU nur, da sie auf der Hardware eingesetzt werden kann, aber für MMBasic eignet sie sich nur bedingt.

Das Mainboard kann entweder alleine, also Stand-Alone (sa), betrieben werden, oder es wird als sogenanntes HAT auf einen Raspberry Pi (RPI) gesteckt. Daher kommt auch die Bezeichnung RPI-sa Controller. Das **ft** am Anfang steht für **fischertechnik**, da ich den Controller hauptsächlich dafür entwickelt und getestet habe.

Programmiert wird der ft-RPI-sa mittels der Programmiersprache BASIC, welche schon sehr früh (ab 1964) in der Geschichte des Programmierens zu finden ist. BASIC ist sehr leicht zu erlernen und Erfolge beim Programmieren stellen sich sehr schnell ein. Das hier verwendete, so genannte MMBasic (Micromite BASIC), besitzt eine weitgehende Kompatibilität zu Microsofts GW-Basic. Dieses wurde nicht von mir, sondern von Geoff Graham entwickelt. Die Anpassung des BASIC Interpreters von Microchips PIC Familie an die NXP S32K1 Familie, sowie die Erweiterung um spezielle Instruktionen für den ft-RPI-sa, wurde von mir vorgenommen.

Wie kommt nun das BASIC Programm in den Controller? Dazu wird entweder ein Computer oder ein Raspberry Pi benötigt. Sämtliche Kommunikation zwischen Controller und Computer geschieht über eine serielle Schnittstelle. Dafür sind 2 Leitungen, eine für das Senden und eine für das Empfangen der Daten, nötig. Die einfachste Art heutzutage diese beiden Leitungen mit einem PC zu verbinden, ist ein so genannter USB zu UART bzw. TTL Konverter. Dieser stellt auf dem Computer eine serielle Komponente, wie z.B. COM1 oder /dev/ttyUSB0 je nach verwendetem Betriebssystem, zur Verfügung, auf welche mit einem simplen Terminalprogramm, wie Putty, TeraTerm unter Windows oder minicom unter Linux, zugegriffen werden kann.

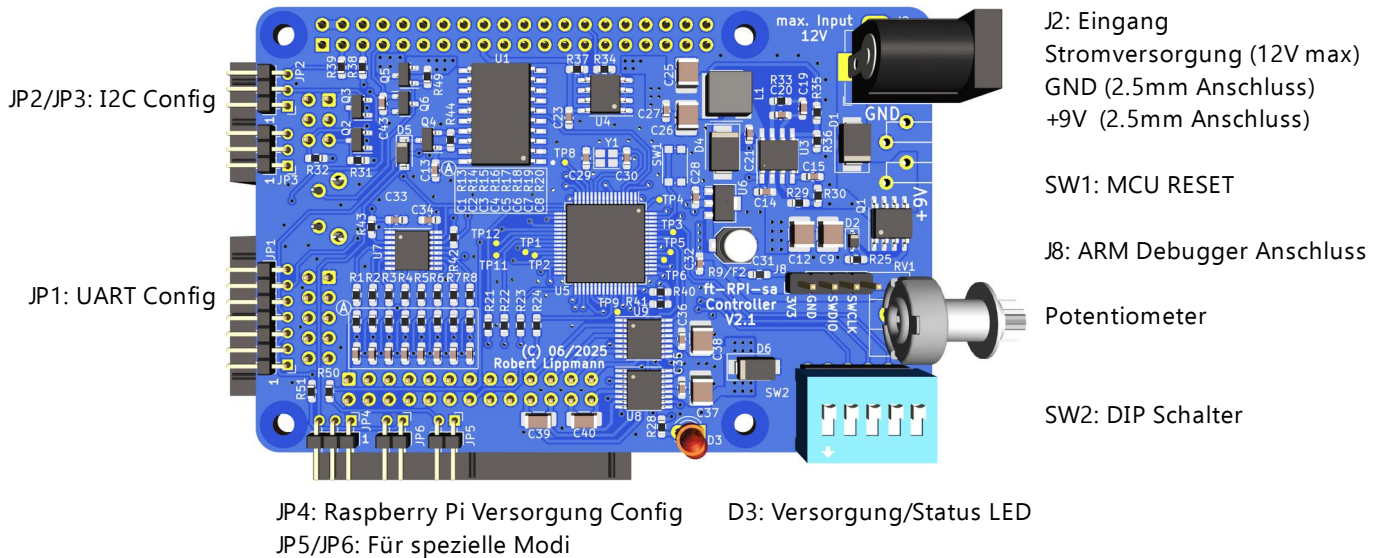
Eine andere, viel bequemere Möglichkeit um eine Verbindung herzustellen, ist die Verwendung eines Raspberry Pis. Dazu wird der ft-RPI-sa einfach auf den RPI gesteckt (HAT) und nach einer korrekten Konfiguration der Schnittstelle steht auch schon die Verbindung. Kein zusätzliches Kabel oder Konverter ist dazu nötig.





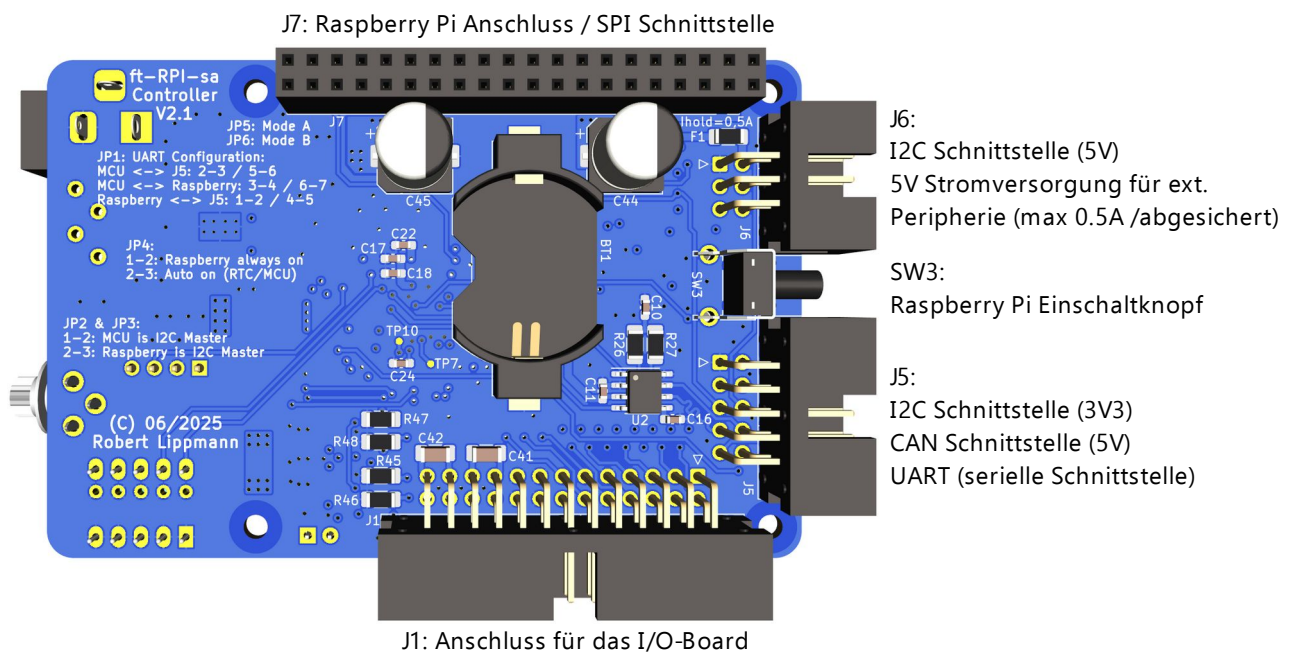
# Die Hardware im Detail

Auf der Rückseite des Mainboards sind schon viele Konfigurationen festgehalten, aber längst nicht alle. Die folgenden Bilder sollen helfen, die Anschlüsse und deren Benutzung zu erklären:



**Ansicht Mainboard von oben**

**Beide Anschlüsse für die Stromversorgung sind gegen Verpolung geschützt!**



**Ansicht Mainboard von unten**

# Konfiguration

## JP1: UART Konfiguration

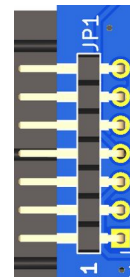
Eine Verbindung zwischen:

2-3 & 5-6 : MCU ↔ J5 (Externer Computer mit Terminalprogramm)

3-4 & 6-7 : MCU ↔ Raspberry Pi (mit Terminalprogramm)

1-2 & 4-5 : Raspberry Pi ↔ J5

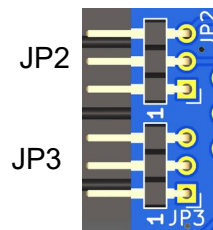
JP1



## JP2 / JP3: I2C Konfiguration

1-2: MCU ist I2C Master (Standard für MMBasic Betrieb)

2-3: Raspberry Pi ist I2C Master



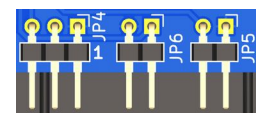
## JP4: Raspberry Pi Versorgung

1-2: Raspberry PI ist immer an

2-3: Raspberry PI ist aus, kann aber über den Taster (SW3),  
die MCU oder einen RTC Alarm eingeschaltet werden

OFF: Raspberry Pi ist immer aus

JP4 JP6 JP5

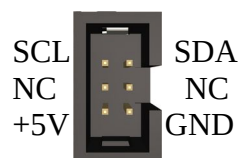


## JP5 / JP6:

Werden benötigt um den Bootloader zu aktivieren und eine neue Firmware einzuspielen.

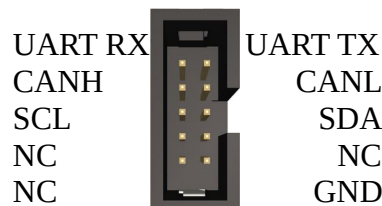
## J5 & J6: Anschlüsse für Peripherie

J6  
5V I2C Anschluss



Die +5V können für externe Peripherie verwendet werden und sind mit 500mA abgesichert.

J5  
I2C/UART/CAN Anschluss



UART und I2C verwenden 3.3V  
CAN verwendet 5V

## J7: Raspberry Pi GPIO Anschluss

Dieser Anschluss ermöglicht ein aufstecken des Controllers auf den Raspberry Pi (HAT = Hardware Attached to the Top).

3v3 Stromversorgung	1		2	5v Stromversorgung
GPIO 2 (SDA)	3		4	5v Stromversorgung
GPIO 3 (SCL)	5		6	Masse (Ground)
GPIO 4 (GPCLK0)	7		8	GPIO 14 (TXD)
Masse (Ground)	9		10	GPIO 15 (RXD)
GPIO 17	11		12	GPIO 18 (PWM0)
GPIO 27	13		14	Masse (Ground)
GPIO 22	15		16	GPIO 23
3v3 Stromversorgung	17		18	GPIO 24
GPIO 10 (MOSI)	19		20	Masse (Ground)
GPIO 9 (MISO)	21		22	GPIO 25
GPIO 11 (SCLK)	23		24	GPIO 8 (CE0)
Masse (Ground)	25		26	GPIO 7 (CE1)
GPIO 0 (ID_SD)	27		28	GPIO 1 (ID_SC)
GPIO 5	29		30	Masse (Ground)
GPIO 6	31		32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33		34	Masse (Ground)
GPIO 19 (MISO)	35		36	GPIO 16
GPIO 26	37		38	GPIO 20 (MOSI)
Masse (Ground)	39		40	GPIO 21 (SCLK)

**Raspberry Pi Belegung (<https://de.pinout.xyz/>)**

Durch das Aufstecken des Controllers auf einen RPI wird die MCU interne LPSPi0 an die SPI1 des Raspberry Pis angeschlossen. Diese Art der synchronen, seriellen Schnittstelle ist verhältnismäßig schnell (bis zu 10MHz) und lässt sich relativ einfach verwenden.

GPIO 19 (SPI1 MISO)	35		36	GPIO 16 (SPI1 CE2)
GPIO 26	37		38	GPIO 20 (SPI1 MOSI)
Masse (Ground)	39		40	GPIO 21 (SPI1 SCLK)

**SPI1 des RPIs an die LPSPi0 der MCU**

Falls die Kommunikation, über die SPI1 des Raspberry Pis, zu dem ft-RPI-sa nicht verwendet wird, oder generell auf einen Raspberry Pi verzichtet wird, kann die MCU interne LPSPi0 für jede andere Art von Peripherie genutzt werden.

Weitere HATs können problemlos auf den ft-RPI-sa Controller gesteckt werden. Dazu musst Du nur darauf achten, dass GPIO4 nicht verwendet werden kann. Dieser wird dazu verwendet, dem ft-RPI-sa mitzuteilen, dass der Raspberry Pi eingeschaltet ist und diesem die Versorgung nicht abschaltet ist.

Wie gerade erwähnt, wird der Raspberry Pi vom ft-RPI-sa Controller versorgt und stellt diesem auch einen Einschaltknopf bereit.

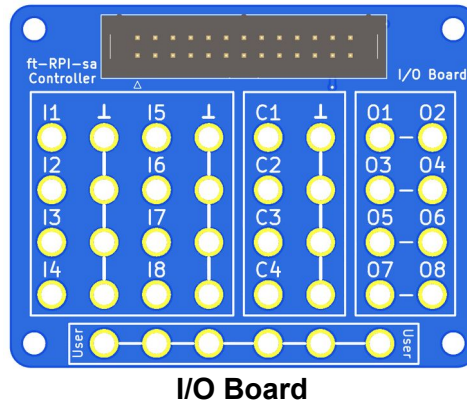
Abhängig von J4 kann der Raspberry Pi sich aber auch automatisch einschalten, entweder per MCU oder per Alarm, gesteuert von der Echtzeituhr auf dem ft-RPI-sa.

Der Raspberry Pi 4 verbraucht im Betrieb bei 5V ca. 600mA, kann aber auch bis zu 1.5A ansteigen, je nach Auslastung des Systems. Da ein sogenannter Schaltregler verwendet wird, um den Raspberry Pi zu versorgen, beträgt der Strom auf der 9V Seite, je nach Auslastung, nur ca. 300mA bzw. ca. 800mA.

Derzeit besteht die einzige Aufgabe des Raspberry Pis darin, den Standard PC als I/O Terminal zu ersetzen - weitere Einsatzmöglichkeiten sind aber schon geplant.

### J1: I/O-Board Anschlüsse

Auf dieser Platine sind 2.5mm Buchsen untergebracht, von denen aus der Controller mit den einzelnen Sensoren und Aktoren der Modelle verbunden wird.



Alle GND (Masse) Verbindungen sind mit einem "⊥" Symbol gekennzeichnet und miteinander verbunden.

Zusätzlich gibt es 8 Eingänge die mit I1 bis I8, sowie 4 Zählereingänge die mit C1 bis C4 gekennzeichnet sind. Neben all den Eingängen gibt es 8 Ausgänge O1 bis O8.

An der unteren Kante ist eine Reihe mit Buchsen versehen, die alle miteinander verbunden sind und die Kennzeichnung "User" haben. Diese 6 Buchsen können als Verteiler verwendet werden.

Die Ausgänge können verschiedenste Aktoren, wie (Schritt-)Motoren, Ventile, Elektromagnete, Lampen etc., mit maximal ~700mA steuern. Jeder der Ausgänge ist individuell steuerbar, für Motoren können jedoch 2 Ausgänge miteinander verbunden werden, um eine vollständige Kontrolle des Motors (Rechtslauf, Linkslauf, Schnelle Bremse oder Ausgleiten) zu bekommen. Dafür sind die zusammengehörigen Ausgänge mit einem Strich dazwischen, auf dem I/O Board gekennzeichnet.

Es kann auch ein Schrittmotor angeschlossen werden. Dazu sind jedoch 4 Ausgänge nötig. In diesem Fall werden die Ausgänge 1 bis 4, sowie 5 bis 8 gruppiert.

Die Eingänge I1 bis I8 sind alle identisch aufgebaut. Jeder der Eingänge verfügt über einen konfigurierbaren 2.7kΩ Pull Widerstand, der wahlweise nach +5V, nach GND oder ganz abgeschaltet werden kann. Mit diesem Pull Widerstand kann ein Eingang, auch wenn er nicht beschaltet ist, auf einen definierten Wert gebracht werden.

Die Zählereingänge C1 bis C4 besitzen keine Pull Möglichkeit und werden von der MCU auf zwei verschiedene Arten realisiert. Zum Einen wird jeder der Zählereingänge jede Millisekunde abgefragt und sobald eine stabile Zustandsänderung von 9V auf 0V stattgefunden hat, der Zähler um 1 weiter gezählt. Somit lassen sich Zustandsänderungen bis maximal 125 mal in der Sekunde erfassen, da für die stabile Zustandsänderung 2x 4 Durchläufe bzw. 8ms benötigt werden. Diese Methode ist dafür geeignet, mechanische Schalter abzufragen, da diese bei der Betätigung mehrere Impulse liefern - das sogenannte Prellen.

Zum Anderen verfügt die MCU über die Möglichkeit, eine Zustandsänderung automatisch zu erfassen ohne Zutun der CPU. Dies erlaubt es, mehr als 100000 Zustandsänderungen pro Sekunde zu verarbeiten. Diese Version der Zähler ist für jede prell-freie Quelle (z.B. Encoder-Motor) zu verwenden.

# Der MMBasic Interpreter

Wie bereits erwähnt, ist MMBasic an das von Microsoft entwickelte GW-Basic angelehnt, jedoch verwendet es keine Zeilennummern. Dies ist ein wesentlicher Vorteil, da es somit nicht so leicht zu dem, bei BASIC Programmen besonders gefürchteten, Spaghetti-Code kommen kann.

## Das Terminal (oder: Die Konsole)

Um den ft-RPI-sa in MMBasic zu programmieren, benutzt man ein Terminal. Ein Terminal ist ein Programm auf dem PC, dass über eine serielle Schnittstelle eine Verbindung zum ft-RPI-sa Controller herstellt. Darüber kann man Kommandos eingeben, um den Controller zu konfigurieren oder zu programmieren, um ein BASIC Programm zu laden oder auch nur das BASIC Programm zu starten.

Das Terminal ist die einzige Möglichkeit um mit dem ft-RPI-sa Controller zu kommunizieren. Es ist daher essentiell diese Verbindung herzustellen. Eine serielle Verbindung besteht aus zwei Leitungen. Eine, oft bezeichnet als Tx, wird zum Senden der Daten und eine weitere Leitung, bezeichnet als Rx, wird zum Empfangen der Daten verwendet. Die Nutzdaten, im ASCII Format, werden umrahmt von einem Start- und einem Stopbit:

8-BIT DATA FORMAT



Die Geschwindigkeit der Übertragung wird in Baud angegeben, was Bits pro Sekunde entspricht. Der ft-RPI-sa Controller verwendet, sobald er eingeschaltet wird, eine Geschwindigkeit von 19200 Baud. Diese kann jedoch, an das verwendete System, angepasst werden. Dazu später mehr. Der Signalpegel auf den Rx/Tx Leitungen schwankt, je nach zu übertragendem Bit-Wert, zwischen 0V und 3.3V, auch TTL Pegel genannt. Um diese beiden Leitungen nun an den PC anzuschließen, ist ein Adapter von USB nach TTL nötig. Dieser ist bereits für kleines Geld im Internet erhältlich.

Dieser USB nach TTL Konverter stellt, sobald dieser am USB Port des Computer eingesteckt wird, ein serielles Interface zur Verfügung.

Je nach verwendetem Betriebssystem ist dies entweder "COMx" (Windows) oder "/dev/ttyUSBx" (Linux). Das x entspricht einer Zahl beginnend bei 1 (Win) oder 0 (Linux) und kann je nach Anzahl der verwendeten Adapter variieren. Auf diese Schnittstellen lässt sich nun mit Hilfe eines entsprechendem Terminalprogramms zugreifen. Unter Windows wäre dies z.B. Putty, TeraTerm und unter Linux kann z.B. minicom verwendet werden.

Um den ft-RPI-sa Controller nun mit dem PC zu verbinden, werden die entsprechenden Signale an dem Anschluss J5 abgegriffen und mit den Anschlüssen des Konverters verbunden:

J5/UART\_RX -> USB2TTL TX sowie

J5/UART\_TX -> USB2TTL RX

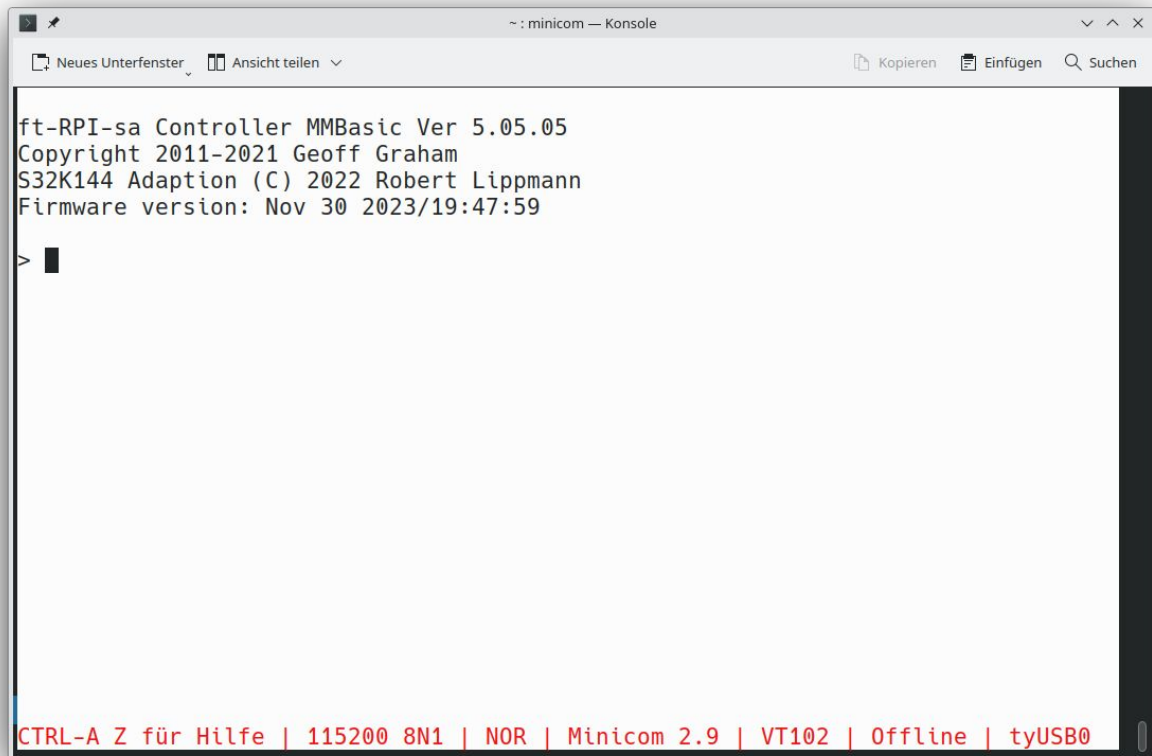
Jetzt kann jemand fragen, wieso nicht RX auf RX bzw. TX auf TX. Ganz einfach: Einen Sender (TX) muss man an einen Empfänger (RX) anschließen.

Eine weitere, bis jetzt noch nicht erwähnte Leitung, muss ebenfalls angeschlossen werden: GND. Dies ist der gemeinsame Bezugspunkt zwischen Computer und ft-RPI-sa Controller.

Sobald alle drei Leitungen verdrahtet sind, kann die erste Kontaktaufnahme gestartet werden und im richtig konfiguriertem Terminalprogramm sollte nach dem Einschalten des ft-RPI-sa Controllers ungefähr folgendes zu sehen sein:







```
~ : minicom — Konsole
Neues Unterfenster Ansicht teilen
Kopieren Einfügen Suchen

ft-RPI-sa Controller MMBasic Ver 5.05.05
Copyright 2011-2021 Geoff Graham
S32K144 Adaption (C) 2022 Robert Lippmann
Firmware version: Nov 30 2023/19:47:59
>

CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0
```

Falls dies nicht der Fall sein sollte: KEINE PANIK!

- Leuchtet die Status-/Power-LED des ft-RPI-sa?
- Sind die beiden Kurzschlussbrücken an JP1 richtig gesetzt?
- Stimmt die eingestellte Baudrate, also Übertragungsgeschwindigkeit, überein?
- Stimmen die Übertragungsparameter (8 Datenbits, 1 Stopbit und keine Parität)?
- Sind die Verbindungen von ft-RPI-sa zu USB2TTL korrekt?
- Funktioniert der USB2TTL Konverter? (Test: Kurzschlussbrücke am USB2TTL auf RX/TX setzen und tippen - jedes gesendete Zeichen sollte im Terminalprogramm zu sehen sein)

Falls ein Raspberry Pi griffbereit ist, kann der ft-RPI-sa Controller auch auf den Raspberry Pi gesteckt werden. In diesem Fall ist kein USB2TTL Converter nötig und nach entsprechendem konfigurieren der Kurzschlussbrücken auf JP1 kann der Raspberry Pi auch schon den PC ersetzen.

Da MMBasic selbst, sowie dessen Befehls- und Funktionsumfang, vom Autor Geoff Graham bereits ordentlich dokumentiert wurde, folgt hier nun eine Kopie seines erstellten Dokuments mit Anpassungen bezüglich ft-RPI-sa spezifischen Befehlen und Funktionen:

# Interacting with MMBasic

The ft-RPI-sa is rather like a self contained computer. You can write programs, run them, edit them and so on, all on the ft-RPI-sa itself. And if your program has an error (as initially all do) you will get a clear message showing you where the fault is located and what the problem is. This is different from other systems where you build your program on a separate computer then download the compiled result to the chip to be run. In that case you do not get any feedback on errors while your program is running... it will just produce crazy results and you will have no idea where or what in your program caused the problem.

This is an important part of working with the ft-RPI-sa – it is much easier and quicker to build a functioning program when you can get immediate feedback and then, with one keystroke, jump into an editor to fix the issue and run the program again.

This interaction with the ft-RPI-sa is done via the console at the command prompt (ie, the greater than symbol (>) on the console). On startup the ft-RPI-sa will issue the command prompt and wait for some command to be entered. It will also return to the command prompt if your program ends or if it generated an error message.

When the command prompt is displayed you have a wide range of commands that you can execute.

Typically your commands would list the program held in flash memory (LIST) or edit it (EDIT) or perhaps set some options (the OPTION command). Most times the command is just RUN which instructs MMBasic to run the program held in flash memory.

Almost any command can be entered at the command prompt and this is often used to test a command to see how it works. A simple example is the PRINT command (more on this in Chapter 3), which you can test by entering the following at the command prompt:

```
PRINT 2 + 2
```

and not surprisingly MMBasic will print out the number 4 before returning to the command prompt. This ability to test a command at the command prompt is very useful when you are learning to program in BASIC, so it would be worthwhile having a ft-RPI-sa handy for the occasional test while you are working through this tutorial.

## Break Key

One useful feature of the ft-RPI-sa is the CTRL-C sequence (hold down the CTRL key then press the C key). This is called the break key or character. When you type this on the console's input it will interrupt whatever the ft-RPI-sa is doing and immediately return control to the command prompt.

This can get you out of all sorts of difficult situations. For example, if you entered the following at the command prompt you would cause MMBasic to enter a continuous loop and appear to be unresponsive.

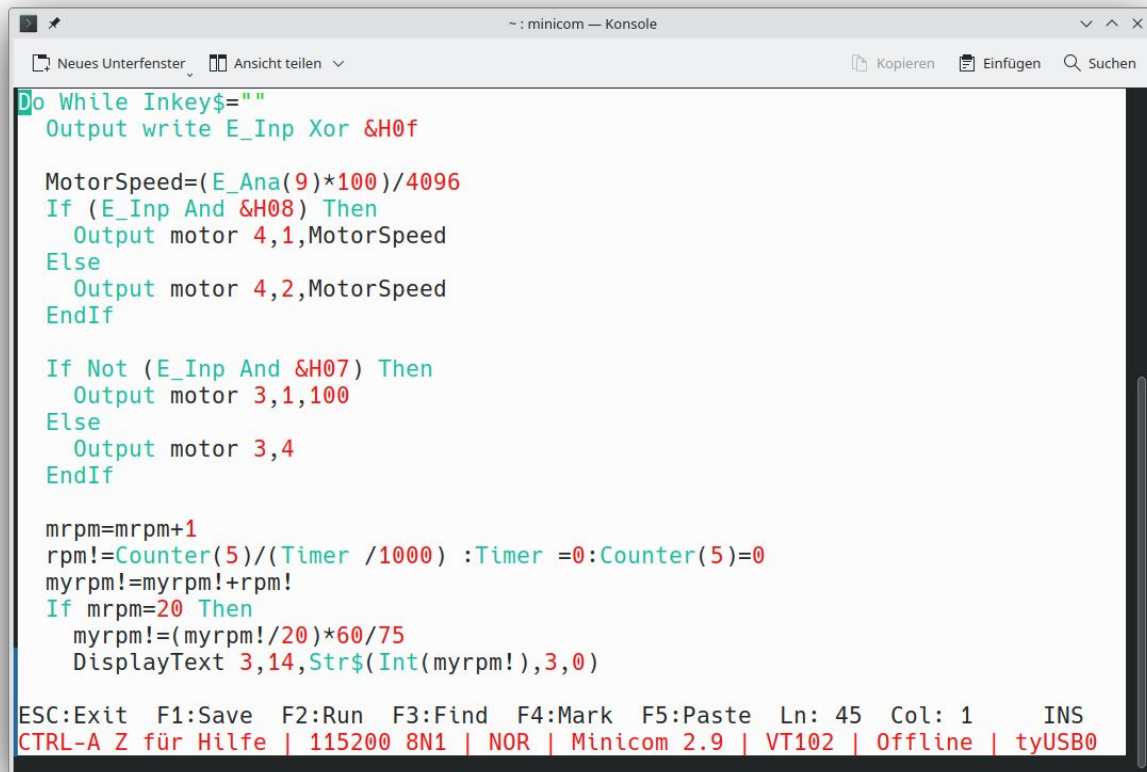
```
DO : LOOP
```

If you have a ft-RPI-sa handy you can try entering this line and you will see that the command prompt does not return because MMBasic is busy spinning in a loop. Then try typing CTRL-C on the console and the ft-RPI-sa will immediately break out of the loop and return to the command prompt.

Remember CTRL-C because it will prove useful at some time in the future.

## The Editor

The ft-RPI-sa has its own built in program editor which can be used to enter programs and correct them when errors are discovered. This screen shot shows the editor in action with colour coded text. Commands are in cyan, comments in yellow, constants in green and so on as shown below.



```
Do While Inkey$=""
Output write E_Inp Xor &H0f

MotorSpeed=(E_Ana(9)*100)/4096
If (E_Inp And &H08) Then
    Output motor 4,1,MotorSpeed
Else
    Output motor 4,2,MotorSpeed
EndIf

If Not (E_Inp And &H07) Then
    Output motor 3,1,100
Else
    Output motor 3,4
EndIf

mrpm=mrpm+1
rpm!=Counter(5)/(Timer /1000) :Timer =0:Counter(5)=0
myrpm!=myrpm!+rpm!
If mrpm=20 Then
    myrpm!=(myrpm!/20)*60/75
    DisplayText 3,14,Str$(Int(myrpm!),3,0)

ESC:Exit  F1:Save  F2:Run  F3:Find  F4:Mark  F5:Paste  Ln: 45  Col: 1  INS
CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0
```

The best way to understand the editor is to try it out. At the command prompt enter the command EDIT and the editor will startup displaying an empty screen with a help line at the bottom of the screen. You can then just type in your program. For example, try typing in:

```
PRINT 1/7
```

Then press the F2 key on your keyboard. This will save the program to memory and run it. This will display the result of dividing 1 by 7.

To change this program use the command EDIT again and you will be taken back into the editor with your program displayed ready for editing.

If you have used an editor like Windows Notepad in the past you will find the operation of this editor familiar. The arrow keys will move your cursor around in the text while the home and end keys will take you to the beginning or end of the line. Page up and page down will do what their titles suggest. The delete key will delete the character at the cursor and backspace will delete the character before the cursor.

About the only unusual key combination is that two home key presses will take you to the start of the program and two end key presses will take you to the end. At the bottom of the screen the status

line will list the various function keys used by the editor and their action. In more details these are:

ESC	This will cause the editor to abandon all changes and return to the command prompt with the program memory unchanged. If you have changed the text you will be asked if you really want to discard your changes.
F1: SAVE	This will save the program to program memory and return to the command prompt.
F2: RUN	This will save the program to program memory and immediately run it.
F3: FIND	This will prompt for the text that you want to search for. When you press enter the cursor will be placed at the start of the first entry found.
SHIFT-F3	After you have used the search function once you can repeatedly search for the same text by pressing SHIFT-F3.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied to the ft-RPI-sa clipboard (see below).

If you pressed the mark key (F4) the editor will change to the *mark mode*. In this mode you can use the arrow keys to mark a section of text which will be highlighted in reverse video. You can then delete, cut or copy the marked text to the ft-RPI-sa's clipboard. In this mode the status line will change to show the functions of the function keys in the mark mode. These keys are:

ESC	Will exit mark mode without changing anything.
F4: CUT	Will copy the marked text to the clipboard and remove it from the text.
F5: COPY	Will just copy the marked text to the clipboard.
DELETE	Will delete the text leaving the clipboard unchanged.

One point to note is that you cannot use the Windows/Linux Copy and Paste function to paste text into the terminal emulator and expect the editor to accept it. This is because the editor has a lot of work to do for each character received and it cannot keep up with a high speed transfer like that. If you do want to copy and paste a program into the terminal emulator you should use the AUTOSAVE command described below.

## Loading a Program from a PC

If you prepare a program on your desktop computer you can transfer it to the ft-RPI-sa using either the AUTOSAVE or XMODEM commands. This requires you to have a terminal emulator running on your desktop machine and connected to the console of the ft-RPI-sa. How to do this was described in the previous chapter.

The AUTOSAVE command puts the ft-RPI-sa into a mode where anything received on the console will be saved to the program memory. This means that you can simply copy the text and paste it into the terminal emulator (eg, minicom) which will send it to the ft-RPI-sa. From the ft-RPI-sa's perspective pasting text into the terminal emulator is the same as if a high speed typist was typing in the program. To terminate the AUTOSAVE command you need to press the Control-Z keys in the terminal emulator and the ft-RPI-sa will save the program to its memory and return to the command prompt.

You can try this if you have a ft-RPI-sa in front of you. At the console type AUTOSAVE and press Enter. You will see nothing on the screen because the ft-RPI-sa is waiting for the input. Copy the program to your computer's clipboard and paste it into your terminal emulator. You should see the program text being echoed back. Enter CTRL-Z and MMBasic will confirm that it has saved the program. Then it is just a case of entering RUN at the command prompt.

The XMODEM command is more sophisticated, it uses the XModem protocol to transfer a BASIC



program file from your personal computer to the ft-RPI-sa including an integrity check which will detect any errors in the transfer. This program file will be saved in the ft-RPI-sa's memory ready to be run.

To start the XModem transfer you need to tell the ft-RPI-sa to receive the file. The command to do this is:

## XMODEM RECEIVE

This instructs the ft-RPI-sa to look for an XModem connection on the console. After running this command you should then instruct your terminal program to send the file using the XModem protocol. In minicom (running on your PC) this is done by using the following menu selection:

CTRL-A → Z → S → XMODEM

This will present a dialog box where you can enter/select the name of the file to be sent and when you click on OK the transfer will start. When the complete file has been received the ft-RPI-sa will save it in program memory and return to the command prompt.

## Setting Options

There are many options that you can set in the ft-RPI-sa and these are all set using the OPTION command. These settings are divided into some that will only last for the duration of the currently running program and therefore are used in a program and others that will be remembered even after

the power has been removed. These are most often entered at the command prompt.

When the ft-RPI-sa is first used the options will have been set to reasonable defaults, so you should not need to change them. But, to give you the flavor of what you can do, the following is a summary of the important settings that are available.

OPTION AUTORUN OFF   ON	Run the program when power is applied
OPTION BASE [0   1]	Set the lower limit of arrays
OPTION BAUDRATE nbr	Change the baudrate of the console
OPTION BREAK nn	Set the character that will break out of a program
OPTION CASE [UPPER   LOWER   TITLE]	Set the format for listing commands
OPTION COLOURCODE ON   OFF	Enable colour coding in the program editor
OPTION DEFAULT [FLOAT   INTEGER   STRING   NONE]	Set the default type of a variable
OPTION DISPLAY lines [,chars]	Size of the display for listing programs
OPTION EXPLICIT	Require that variables be properly declared
OPTION RESET	Reset all options to their defaults
OPTION TAB [2   4   8]	Set the size of tabs

# Programming Fundamentals

The ft-RPI-sa is programmed using the BASIC programming language. The ft-RPI-sa version of BASIC is called MMBasic (short for the origin MicroMite BASIC) which is loosely based on the Microsoft BASIC interpreter that was popular years ago.

The BASIC language was introduced in 1964 by Dartmouth College in the USA as a computer language for teaching programming and accordingly it is easy to use and learn. At the same time, it has proved to be a competent and powerful programming language and as a result it became very popular in the 70s and 80s. Even today some large commercial data systems are still written in the BASIC language (primarily Pick Basic).

For the ft-RPI-sa the greatest advantage of BASIC is its ease of use. Some more modern languages such as C and C++ can be truly mind bending but with BASIC you can start with a one line program and get something sensible out of it. MMBasic is also powerful in that you can control the I/O pins on the ft-RPI-sa or communicate with other chips using a range of built-in communications protocols.

The only significant downside to using a BASIC interpreter is that it is not as fast as a fully compiled language like C or C++. However, on a fast microcontroller such as the S32K144W MMBasic will execute each command in an average of 50µs or faster. This speed is suitable for most applications - for example, it is fast enough to respond to a high revving engine and deal with signals such as ignition triggers.

## Structure of a BASIC Program

A BASIC program starts at the first line and continues until it runs off the end of the program or hits an END command - at which point MMBasic will display the command prompt (>) on the console and wait for something to be entered.

A program consists of a number of statements or commands, each of which will cause the BASIC interpreter to do something (the words statement and command generally mean the same and are used interchangeable in this book). Normally each statement is on its own line but you can have multiple statements in the one line separated by the colon character (:). For example;

```
A = 24.6 : PRINT A
```

Each line can start with a line number. Line numbers were mandatory in the early BASIC interpreters however modern implementations (such as MMBasic) do not need them. You can still use them if you wish but they have no benefit and generally just clutter up your programs. This is an example of a program that uses line numbers:

```
50 A = 24.6  
60 PRINT A
```

A line can also start with a label which can be used as the target for a program jump using the GOTO command. This will be explained in more detail when we cover the GOTO command but this is an example (the label name is JmpBack):

```
JmpBack: A = A + 1  
PRINT A  
GOTO JmpBack
```

## The PRINT Command

There are a number of common commands that are fundamental and we will cover them in this

chapter but arguably the most useful is the PRINT command. Its job is simple; to print something on the console. This is mostly used to tell you how your program is running and can consist of something simple such as "Pump running" or "Total Flow: 23 litres".

PRINT is also useful when you are tracing a fault in your program; you can use it to print out the values of variables, I/O pins and display messages at key stages in the execution of the program. In its simplest form the command will just print whatever is on its command line. So, for example:

```
PRINT 54
```

Will display on the console the number 54 followed by a new line.

The data to be printed can be an expression, which means something to be calculated. We will cover expressions in more detail later but as an example the following:

```
> PRINT 3/21
0.142857
>
```

would calculate the result of three divided by twenty one and display it. Note that the greater than symbol (>) is the command prompt produced by MMBasic – you do not type that in.

Other examples of the PRINT command include:

```
> PRINT "Hello World"
Hello World
> PRINT (999 + 1) / 5
200
>
```

You can try these out at the command prompt.

The PRINT command will also work with multiple values at the same time, for example:

```
> PRINT "The amount is" 345 " and the second amount is" 456
The amount is 345 and the second amount is 456
>
```

Normally each value is separated by a space character as shown in the previous example but you can also separate values with a comma (,). The comma will cause a tab to be inserted between the two values. In MMBasic tabs in the PRINT command are eight characters apart. To illustrate tabbing the following command prints a tabbed list of numbers:

```
> PRINT 12, 34, 9.4, 1000
12 34 9.4 1000
>
```

Note that there is a space printed before each number. This space is a place holder for the minus symbol (-) in case the value is negative. Notice the difference with the number 12 in this example:

```
> PRINT -12, 34, -9.4, 1000
-12 34 -9.4 1000
>
```

The print statement can be terminated with a semicolon (;). This will prevent the PRINT command from moving to a new line when it completes printing all the text. For example:

```
PRINT "This will be";
```

```
PRINT " printed on a single line."
```

Will result in this output:

```
This will be printed on a single line.
```

The message would be look like this without the semicolon at the end of the first line:

```
This will be  
printed on a single line.
```

## Variables

Before we go much further we need to define what a "variable" is as they are fundamental to the operation of the BASIC language (in fact, any programming language). A variable is simply a place to store an item of data (ie, its "value"). This value can be changed as the program runs which

why it is called a "variable".

Variables in MMBasic can be one of three types. The most common is floating point and this is automatically assumed if the type of the variable is not specified. The other two types are integer and string and we will cover them later. A floating point number is an ordinary number which can contain a decimal point. For example 3.45 or -0.023 or 100.00 are all floating point numbers.

A variable can be used to store a number and it can then be used in the same manner as the number

itself, in which case it will represent the value of the last number assigned to it.

As a simple example:

```
A = 3  
B = 4  
PRINT A + B
```

will display the number 7. In this case both A and B are variables and MMBasic used their current values in the PRINT statement. MMBasic will automatically create a variable when it first encounters it so the statement A = 3 both created a floating point variable (the default type) with the name of A and then it assigned the value of 3 to it.

The name of a variable must start with a letter while the remainder of the name can use letters, numbers, the underscore or the full stop (or period) characters. The name can be up to 32 characters

long and the case (ie, capitals or not) is not important. Here are some examples:

```
Total_Count  
ForeColour  
temp3  
count
```

You can change the value of a variable anywhere in your program by using the assignment command, ie:

```
variable = expression
```

For example:



```
temp3 = 24.6
count = 5
CTemp = (FTemp - 32) * 0.5556
```

In the last example both CTemp and FTemp are variables and this line converts the value of FTemp (in degrees Fahrenheit) to degrees Celsius and stores the result in the variable Ctemp.

## Expressions

We have met the term 'expression' before in this tutorial and in programming it has a specific meaning. It is a formula which can be resolved by the BASIC interpreter to a single number or value.

MMBasic will evaluate a mathematical expression using the same rules that we all learnt at school. For example, multiplication and division are performed first followed by addition and subtraction. These are called the rules of precedence and are fully spelt out in the manual (around page 52). This means that  $2 + 3 * 6$  will resolve to 20, so will  $5 * 4$  and also  $10 + 4 * 3 - 2$ .

If you want to force the interpreter to evaluate parts of the expression first you can surround that part of the expression with brackets. For example,  $(10 + 4) * (3 - 2)$  will resolve to 14 not 20 as would have been the case if the brackets were not used. Using brackets does not appreciably slow down the program so you should use them liberally if there is a chance that MMBasic will misinterpret your expression.

As pointed out earlier, you can use variables in an expression exactly the same as straight numbers.

For example, this will increment the value of the variable temp by one:

```
temp = temp + 1
```

You can also use functions in expressions. These are special operations provided by MMBasic, for example to calculate trigonometric values. As an example, the following will print the length of the hypotenuse of a right angled triangle using the SQR() function which returns the square root of a number (a and b are variables holding the lengths of the other sides):

```
PRINT SQR(a * a + b * b)
```

MMBasic will first evaluate this expression by multiplying a by a, then multiplying b by b, then adding the results together. The resulting number is then passed to the SQR() function which will calculate the square root of that number and return it for the PRINT command to display.

Some other mathematical functions provided by MMBasic include:

SIN(r) – the sine of r

COS(r) – the cosine of r

TAN(r) – the tangent of r

There are many more functions available to you and they are all listed in the User Manual.

Note that in the above functions the value passed to the function (ie, 'r') is the angle in radians. In MMBasic you can use the function RAD(d) to convert an angle from degrees to radians ('d' is the angle in degrees).

Another feature of BASIC is that you can nest function calls within each other. For example, given the angle in degrees (ie, 'd') the sine of that angle can be found with this expression:

```
PRINT SIN(RAD(d))
```

In this case MMBasic will first take the value of d and convert it to radians using the RAD() function. The output of this function then becomes the input to the SIN() function.

## The IF statement

Making decisions is at the core of most computer programs and in BASIC that is usually done with the IF statement. This is written almost like an English sentence:

```
IF condition THEN action
```

The condition is usually a comparison such as equals, less than, more than, etc. For example:

```
IF Temp < 25 THEN PRINT "Cold"
```

Temp would be a variable holding the current temperature (in °C).

There are a range of tests that you can make:

=	equals	<>	not equal
<	less than	<=	less than or equals
>	greater than	>=	greater than or equals

You can also add an ELSE clause which will be executed if the initial condition tested false. For example, this will execute different actions when the temperature is under 25 or 25 or more:

```
IF Temp < 25 THEN PRINT "Cold" ELSE PRINT "Hot"
```

The previous examples all used single line IF statements but you can also have multiline IF statements. They look like this:

```
IF condition THEN
    TrueActions
ENDIF
```

Or, if you also want to take some actions if false:

```
IF condition THEN
    TrueActions
ELSE
    FalseActions
ENDIF
```

Unlike the single line IF statement you can have many true actions with each on their own line and similarly many false actions. Generally the single line IF statement is handy if you have a simple action that needs to be taken while the multiline version is much easier to understand if the actions are numerous and more complicated.

An example of a multiline IF statement with more than one action is:

```
IF Temp < 25 THEN
    PRINT "Cold"
    TurnRedLightOff
ELSE
    PRINT "Hot"
```

```
        TurnRedLightOn
    ENDIF
```

Note that in the above example each action is indented to show what part of the IF structure it belongs to. Indenting is not mandatory but it makes a program much easier to understand for someone who is not familiar with it and therefore it is highly recommended. You will find that this tutorial uses indenting in all examples for this reason.

In a multiline IF statement you can make additional tests using the ELSE IF command. This is best explained by using an example:

```
    IF Temp < 0 THEN
        PRINT "Freezing"
    ELSE IF Temp < 25 THEN
        PRINT "Cold"
    ELSE IF Temp < 40 THEN
        PRINT "Warm"
    ELSE
        PRINT "Hot"
    ENDIF
```

The ELSE IF can use the same tests as an ordinary IF (ie, <, <=, etc) but that test will only be made

if the preceding test was false. So, for example, you will only get the message *Warm* if Temp < 0 failed, and Temp < 25 failed but Temp < 40 was true. The final ELSE will catch the case where all the tests were false.

An expression like Temp < 25 is evaluated by MMBasic as either true or false with true having a value of one and false zero. You can see this if you entered the following at the console:

```
PRINT 30 > 20
```

MMBasic will print 1 meaning that the value is true and similarly the following will print 0 meaning that the expression evaluated to false.

```
PRINT 30 < 20
```

The IF statement does not really care about what the condition actually is, it just evaluates the condition and if the result is zero it will take that as false and if non zero it will take it as true. This allows for some handy shortcuts. For example, if SwitchOn is a variable that is true (non zero) when some switch is turned on the following can be used to make a decision based on that value:  
IF SwitchOn THEN ...*do something*...

FOR Loops

Another common requirement in programming is repeating a set of actions. For instance, you might want to step through all seven days in the week and perform the same function for each day. BASIC provides the FOR loop construct for this type of job and it works like this:

```
FOR day = 1 TO 7
```

*Do something based on the value of 'day'*

```
NEXT day
```

This starts by creating the variable day and assigning the value of 1 to it. The program then will execute the following statements until it comes to the NEXT statement. This tells the BASIC

interpreter to increment the value of day, go back to the previous FOR statement and re-execute the following statements a second time. This will continue looping around until the value of day exceeds 7 and the program will then exit the loop and continue with the statements following the NEXT statement.

As a simple example, you can print the numbers from one to ten like this:

```
FOR nbr = 1 TO 10
  PRINT nbr;;
NEXT nbr
```

The comma at the end of the PRINT statement tells the interpreter to tab to the next tab column after printing the number while the semicolon will leave the cursor on this line rather than automatically moving to the next line. As a result the numbers will be printed in neat columns across the page. Try it on a ft-RPI-sa, use the EDIT command to enter the short program listed above then press the F2 key to save and run it.

The FOR loop also has a couple of extra tricks up it sleeve. You can change the amount that the variable is incremented by using the STEP keyword. So, for example, the following will print just the odd numbers:

```
FOR nbr = 1 TO 10 STEP 2
  PRINT nbr;;
NEXT nbr
```

The value of the step (or increment value) defaults to one if the STEP keyword is not used but you can set it to whatever number you want.

When MMBasic is incrementing the variable it checks to see if the variable has exceeded the TO value and, if it has, it will exit from the loop. So, in the above example, the value of nbr will reach nine and it will be printed but on the next loop nbr will be eleven and at that point execution will leave the loop. This test is also applied at the start of the loop (ie, if in the beginning the value of the variable exceeds the TO value the loop will never be executed, not even once).

By setting the STEP value to a negative number you can use the FOR loop to step down from a high number to low. For example, the following will print the numbers from 10 to 1 in reverse:

```
FOR nbr = 10 TO 1 STEP -1
  PRINT nbr;;
NEXT nbr
```

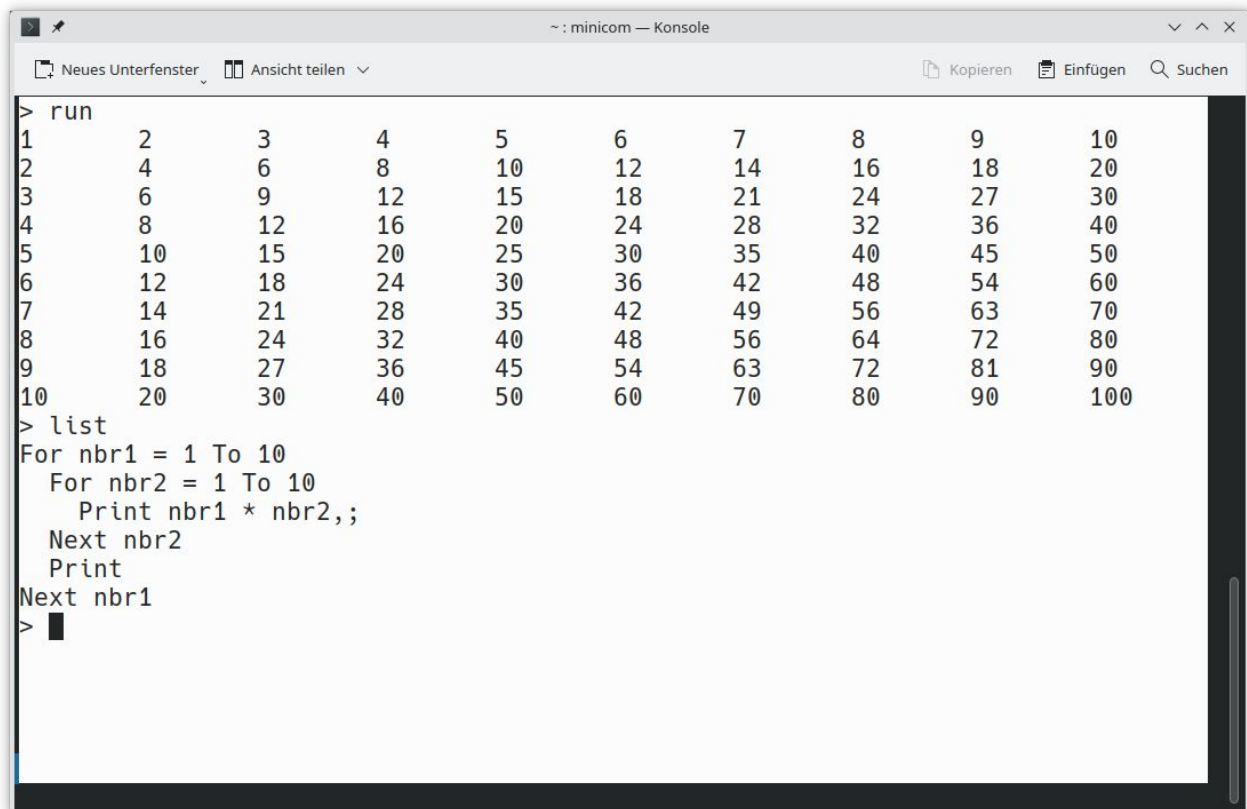
## Multiplication Table

To further illustrate how loops work and how useful they can be, the following short program will use the FOR loop to print out the multiplication table that we all learnt at school. The program for this is not complicated:

```
FOR nbr1 = 1 to 10
  FOR nbr2 = 1 to 10
    PRINT nbr1 * nbr2;;
  NEXT nbr2
  PRINT
NEXT nbr1
```

The output will be similar to the screen grab below, which also shows a listing of the program.





```
> run
1      2      3      4      5      6      7      8      9      10
2      4      6      8     10     12     14     16     18     20
3      6      9     12     15     18     21     24     27     30
4      8     12     16     20     24     28     32     36     40
5     10     15     20     25     30     35     40     45     50
6     12     18     24     30     36     42     48     54     60
7     14     21     28     35     42     49     56     63     70
8     16     24     32     40     48     56     64     72     80
9     18     27     36     45     54     63     72     81     90
10    20     30     40     50     60     70     80     90    100
> list
For n1 = 1 To 10
  For n2 = 1 To 10
    Print n1 * n2,;
  Next n2
  Print
Next n1
>
```

You need to work through the logic of this example line by line to understand what it is doing. Essentially it consists of one loop inside another. The inner loop, which increments the variable `n2`, prints one horizontal line of the table. When this loop has finished it will execute the following `PRINT` command which has nothing to print - so it will simply output a new line (ie, terminate the line printed by the inner loop).

The program will then execute another iteration of the outer loop by incrementing `n1` and re-executing the inner loop again. Finally, when the outer loop is exhausted (when `n1` exceeds 10) the program will reach the end and terminate.

One last point, you can omit the variable name from the `NEXT` statement and MMBasic will guess which variable you are referring to. However, it is good practice to include the name to make it easier for someone else who is reading the program. You can also terminate multiple loops using a comma separated list of variables in the `NEXT` statement. For example:

```
FOR var1 = 1 TO 5
  FOR var2 = 10 to 13
    PRINT var1 * var2
  NEXT var1, var2
```

## DO Loops

Another method of looping is the `DO...LOOP` structure which looks like this:

```
DO WHILE condition
  statement
  statement
LOOP
```

This will start by testing the condition and if it is true the statements will be executed until the LOOP command is reached, at which point the condition will be tested again and if it is still true the loop will execute again. The 'condition' is the same as in the IF command (ie,  $X < Y$ ). For example, the following will keep printing the word "Hello" on the console for 4 seconds then stop:

```
Timer = 0
DO WHILE Timer < 4000
    PRINT "Hello"
LOOP
```

Note that Timer is a function within MMBasic which will return the time in milliseconds since the timer was reset. A reset is done by assigning zero to Timer (as done above) or when powering up the ft-RPI-sa. We will cover the timer in more detail later.

A variation on the DO-LOOP structure is the following:

```
DO
    statement
    statement
LOOP UNTIL condition
```

In this arrangement the loop is first executed once, the condition is then tested and if the condition is false, the loop will be repeatedly executed until the condition becomes true. Note that the test in LOOP UNTIL is the inverse of DO WHILE.

For example, similar to the previous example, the following will also print "Hello" for four seconds:

```
Timer = 0
DO
    PRINT "Hello"
LOOP UNTIL Timer >= 4000
```

Both forms of the DO-LOOP essentially do the same thing, so you can use whatever structure fits with the logic that you wish to implement.

Finally, it is possible to have a DO Loop that has no conditions at all – ie,

```
DO
    statement
    statement
LOOP
```

This construct will continue looping forever and you, as the programmer, will need to provide a way to explicitly exit the loop (the EXIT DO command will do this). For example:

```
Timer = 0
DO
    PRINT "Hello"
    IF Timer >= 4000 THEN EXIT DO
LOOP
```

## Console Input

There are times where you would like to use the ft-RPI-sa as a straight computer with all of its

input coming from the console and its output going to the same place. For that to work you need to capture keystrokes from the console and this can be done with the INPUT command. In its simplest form the command is:

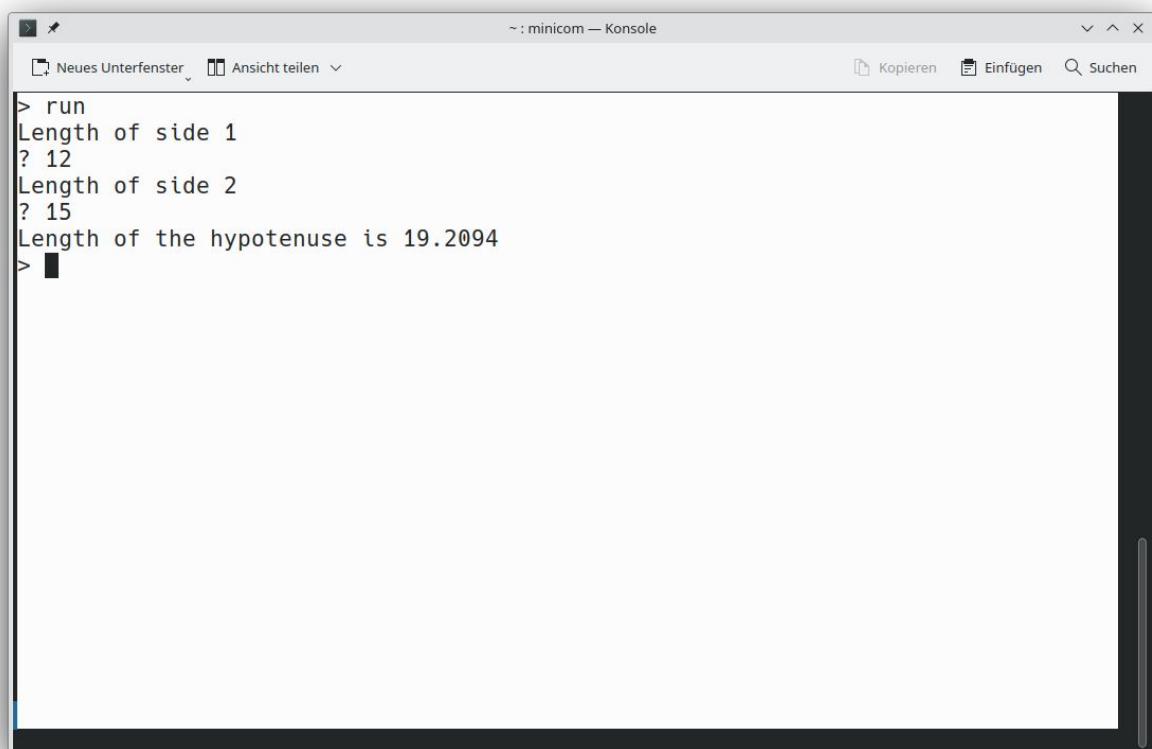
```
INPUT var
```

This command will print a question mark on the console's screen and wait for a number to be entered followed by the Enter key. That number will then be assigned to the variable var.

For example, the following program extends the expression for finding the hypotenuse of a triangle by allowing the user to enter the lengths of the other sides from the console.

```
PRINT "Length of side 1"  
INPUT a  
PRINT "Length of side 2"  
INPUT b  
PRINT "Length of the hypotenuse is " SQR(a * a + b * b)
```

This is a screen capture of a typical session:



```
> run  
Length of side 1  
? 12  
Length of side 2  
? 15  
Length of the hypotenuse is 19.2094  
> █
```

The INPUT command can also print your prompt for you, so that you do not need a separate PRINT command. For example, this will work the same as the above program:

```
INPUT "Length of side 1"; a
```

```
INPUT "Length of side 2"; b
```

```
PRINT "Length of the hypotenuse is " SQR(a * a + b * b)
```

Finally, the INPUT command will allow you to input a series of numbers separated by commas with each number being saved in different variables. For example:

```
INPUT "Enter the length of the two sides: ", a, b
```

```
PRINT "Length of the hypotenuse is " SQR(a * a + b * b)
```

If the user entered 12,15 the number 12 would be saved in the variable a and 15 in b.

Another method of getting input from the console is the LINE INPUT command. This will get the whole line as typed by the user and allocate it to a string variable. Like the INPUT command you can also specify a prompt. This is a simple example:

```
LINE INPUT "What is your name? ", s$
```

```
PRINT "Hello " s$
```

We will cover string variables in the next chapter but for the moment you can think of them as a variable that holds a sequence of one or more characters. If you ran the above program and typed in

John when prompted the program would respond with Hello John.

Sometimes you do not want to wait for the user to hit the enter key, you want each character as it is

typed in. This can be done with the INKEY\$ function which will return the value of the character as a string (also covered in the next chapter).



# GOTO and Labels

One method of controlling the flow of the program is the GOTO command. This essentially tells MMBasic to jump to another part of the program and continue executing from there. The target of the GOTO is a label and this needs to be explained first.

A label is an identifier that marks part of the program. It must be the first thing on the line and it must be terminated with the colon (:) character. The name that you use can be up to 32 characters long and must follow the same rules for a variable's name. For example, in the following program line LoopBack is a label:

```
LoopBack: a = a + 1
```

When you use the GOTO command to jump to that particular part of the program you would use the command like this:

```
GOTO LoopBack
```

To put all this into context the following program will print out all the numbers from 1 to 10:

```
z = 0
LoopBack: z = z + 1
PRINT z
IF z < 10 THEN GOTO LoopBack
```

The program starts by setting the variable z to zero then incrementing it to 1 in the next line. The value of z is printed and then tested to see if it is less than 10. If it is less than 10 the program execution will jump back to the label LoopBack where the process will repeat. Eventually the value of z will be more than 10 and the program will run off the end and terminate.

Note that a FOR loop can do the same thing (and is simpler) so this example is purely designed to illustrate what the GOTO command can do.

In the past the GOTO command gained a bad reputation. This is because using GOTOs it is possible to create a program that continuously jumps from one point to another (often referred to as

"spaghetti code") and that type of program is almost impossible for another programmer to understand. With constructs like the multiline IF statements the need for the GOTO statement has been reduced and it should be used only when there is no other way of changing the program's flow.

## Testing for Prime Numbers

The following is a simple program which brings together many of the programming features previously discussed.

```
DO
  InpErr:
  PRINT
  INPUT "Enter a number: "; a
  IF a < 2 THEN
    PRINT "Number must be equal or greater than 2"
    GOTO InpErr
```

```

ENDIF
Divs = 0
FOR x = 2 TO SQR(a)
    r = a/x
    IF r = FIX(r) THEN Divs = Divs + 1
NEXT x
PRINT a " is ";
IF Divs > 0 THEN PRINT "not ";
PRINT "a prime number."
LOOP

```

This will first prompt (on the console) for a number and, when it has been entered, it will test if that number is a prime number or not and display a suitable message.

It starts with a DO Loop that does not have a condition – so it will continue looping forever. This is what we want. It means that when the user has entered a number, it will report if it is a prime number or not and then loop around and ask for another number. The way that the user can exit the program (if they wanted to) is by typing the break character (normally CTRL-C).

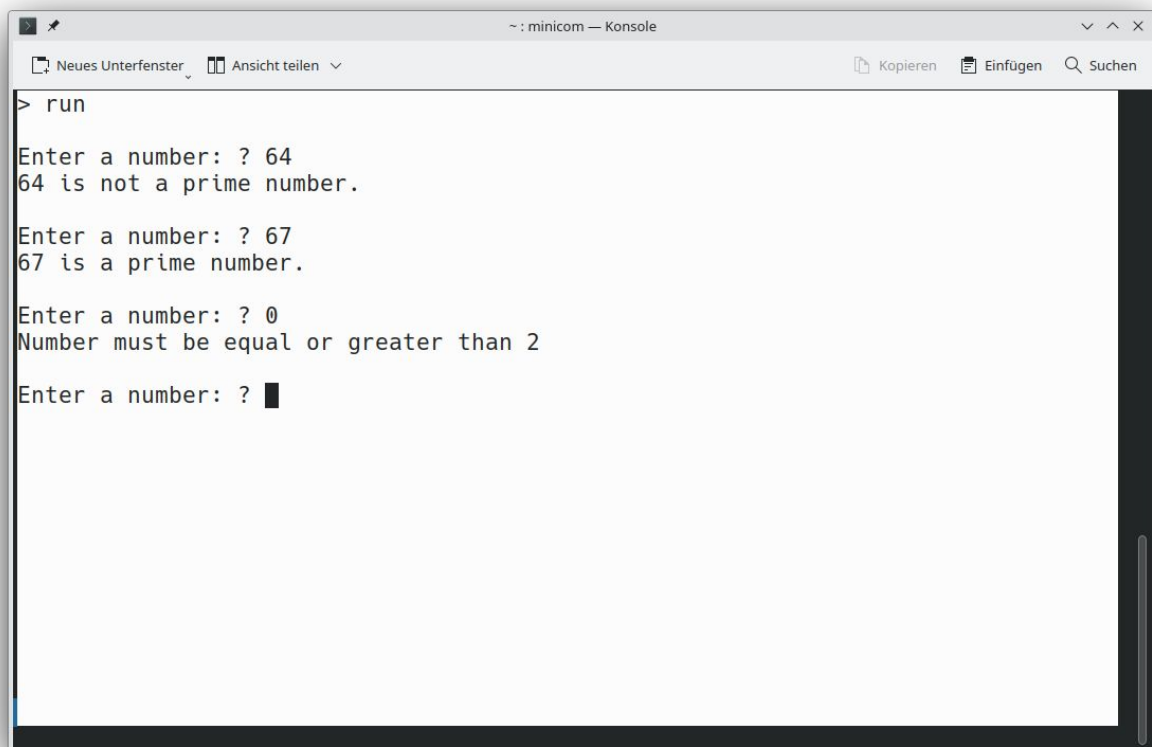
The program then prints a prompt for the user which is terminated with a semicolon character. This means that the cursor is left at the end of the prompt for the INPUT command which will get the number and store it in the variable a.

Following this the number is tested. If it is less than 2 an error message will be printed and the program will jump backwards and ask for the number again.

We are now ready to test if the number is a prime number. The program uses a FOR loop to step through the possible divisors testing if each one can divide evenly into the entered number. Each time it does the program will increment the variable Divs. Note that the test is done with the function FIX(r) which simply strips off any digits after the decimal point. So, the condition  $r = \text{FIX}(r)$  will be true if r is an integer (ie, has no digits after the decimal point).

Finally, the program will construct the message for the user. The key part is that if the variable Divs is greater than zero it means that one or more numbers were found that could divide evenly into the test number. In that case the IF statement inserts the word "not" into the output message. For example, if the entered number was 21 the user will see this response:  
21 is not a prime number.

This is the result of running the program and some of the output:



```
> run

Enter a number: ? 64
64 is not a prime number.

Enter a number: ? 67
67 is a prime number.

Enter a number: ? 0
Number must be equal or greater than 2

Enter a number: ? █
```

You can test this program by using the editor (the EDIT command) to enter it.

Using your newly learned skills you could then have a shot at making it more efficient. For example, because the program counts how many times a number can be divided into the test number it takes a lot longer than it should to detect a non prime number. The program would run much more efficiently if it jumped out of the FOR loop at the first number that divided evenly. You could use the GOTO command to do this or you could use the command EXIT FOR – that would cause the FOR loop to terminate immediately.

Other efficiencies include only testing the division with odd numbers (by using an initial test for an even number then starting the FOR loop at 3 and using STEP 2) or by only using prime numbers for the test (that would be much more complicated).

## Saving the Program

When you entered the program then saved it you might wonder where your program has actually been saved to. The answer is that it was automatically written into the dataflash memory of the S32K1xx chip. In fact, if you save a very large program you might see a delay of a second or two which is the time needed by MMBasic to transfer a large amount of data into the dataflash memory.

Flash memory is non volatile which means that it will retain its contents when the power is removed. This might not be important for a program that checks for prime numbers but if you have

programmed the ft-RPI-sa to be the brains in your burglar alarm you will not want it to lose the program during a blackout.

The command MEMORY will report on how much memory was used by the program. With the above program it will display something like this:

Flash:

1K ( 1%) Program (19 lines)

59K (99%) Free

RAM:

1K ( 1%) 4 Variables

0K ( 0%) General

49K (99%) Free

As you can see, the program used little memory. This is another advantage of the ft-RPI-sa; the relatively huge memory space means that you can create large and complex programs and still run them on this small and inexpensive chip.

# Advanced BASIC Programming

In the previous chapter we covered the fundamentals of BASIC programming, enough to write a small program to do a simple job. But BASIC has additional features that become important when you are constructing a more complex program and we will cover these in this chapter. As stressed before, this tutorial is not intended as a comprehensive manual and as such there are many more specialized features that are not covered here. To discover these it is recommended that you download the *Micromite User Manual* from: <http://geoffg.net/micromite.html>

## Utility Commands

Before we go much further we should discuss some of the utility commands and features of MMBasic that help you manage and run your program.

The first is the comment which is any text that follows the single quote character ('). A comment can be placed anywhere and extends to the end of the line. If MMBasic runs into a comment it will just skip to the end of it (ie, it does not take any action regarding a comment).

Comments should be used to explain non obvious parts of the program and generally inform someone who is not familiar with the program how it works and what it is trying to do. Remember that after only a few months a program that you have written will have faded from your mind and will look strange when you pick it up again. For this reason you will thank yourself later if you use plenty of comments.

The following are some examples of comments:

```
' calculate the hypotenuse  
PRINT SQR(a * a + b * b)
```

or

```
INPUT var ' get the temperature
```

We have covered the EDIT command which will run the internal ft-RPI-sa program editor. Other useful commands are LIST which will list your program on the console (pausing every 24 lines) and the RUN command which will start your program running.

If you want to completely clear the program in memory you can use the NEW command which will erase everything leaving you with the maximum free program memory. The CLEAR command will do the same for any variables (ie, delete them and recover the memory). As shown before, the MEMORY command will list how much memory is currently being used.

The TRACE command is useful if you are trying to work out what your program is doing wrong. TRACE ON will cause MMBasic to list the line number of each statement as it is executed and this can help you trace the program flow. TRACE OFF will stop this feature. This command can also be embedded in your program so you can turn on tracing for short sections of code if you wish. Finally there is the OPTION command as mentioned in chapter 2. This takes many forms and using

it you can change many settings within MMBasic including how programs are listed, the attached devices (such as LCD panels), how your program will be run and much more.



# Arrays

An array is something which you will probably not think of as useful at first glance but when you do need to use them you will find them very handy indeed.

An array is best thought of as a row of letterboxes for a block of units or condos as shown on the right. The letterboxes are all located at the same address and each box represents a unit or condo at that address. You can place a letter in the box for unit one, or unit two, etc.



Similarly an array in BASIC is a single variable with multiple sub units (called *elements* in BASIC) which are numbered. You can place data in element one, or element two, etc. In BASIC an array is created by the DIM command, for example:

```
DIM numarr(300)
```

This created an array with the name of numarr and containing 301 elements (think of them as letterboxes). By default an array will start from zero so this is why there is an extra element making the total 301. To specify a specific element in the array (ie, a specific letterbox) you use an index which is simply the number of the array element that you wish to access. For example, if you want to set element number 100 in this array to (say) the number 876, you would do it this way:

```
numarr(100) = 876
```

Normally the index to an array is not a constant number as shown here but a variable which can be changed to access different array elements.

As an example of how you might use an array, consider the case where you would like to record the

temperature for each day of the year and, at the end of the year, calculate the overall average. You could use ordinary variables to record the temperature for each day but you would need 365 of them

and that would make your program unwieldy indeed. Instead, you could define an array to hold the values like this:

```
DIM days(365)
```

Every day you would need to save the temperature in the correct location in the array. If the number

of the day in the year was held in the variable doy and the maximum temperature was held in the variable maxtemp you would save the reading like this:

```
days(doy) = maxtemp
```

At the end of the year it would be simple to calculate the average for the year:

```
total = 0
FOR i = 1 to 365
  total = total + days(i)
NEXT i
PRINT "Average is:" total / 365
```

This is much easier than adding up and averaging 365 individual variables.

The above array was single dimensioned but you can have multiple dimensions. Reverting to our analogy of letterboxes, an array with two dimensions could be thought of as a block of flats with multiple floors.

A block could have a row of four letter boxes for level one, another row of four boxes for level two, and so on. To place a letter in a letterbox you need to specify the floor number and the unit number on that floor.

In BASIC the array element is specified using two indices separated by a comma. For example:

LetterBox(floor, unit)

As a practical example, assume that you needed to record the maximum temperature for each day over five years. To do this you could dimension the array as follows:

DIM days(365, 5)

The first index is the day in the year and the second is a number representing the year. If you wanted to set day 100 in year 3 to 24 degrees you would do it like this:

days(100, 3) = 24

In MMBasic you can have up to eight dimensions and the maximum size of an array is only limited by the amount of free RAM that is available in the ft-RPI-sa.

## Integers

So far all the numbers and variables that we have been using have been floating point. As explained before, floating point is handy because it will track digits after the decimal point and when you use division it will return a sensible result. So, if you just want to get things done and are not concerned with the details you should stick to floating point.

However, the limitation of floating point is that it stores numbers as an approximation with an accuracy of only 7 or 8 digits in the processor.

For example, if you stored the number 1234.56789 in a floating point variable on the ft-RPI-sa then printed it out you will find that the value stored in the variable is actually 1234.57.

The limit on the accuracy of floating point can cause confusion for new programmers. For example, a beginner might write the following program for the ft-RPI-sa and be surprised when the value of F does not change.

```
F = 1234.5
PRINT F
F = F + 0.0001
PRINT F
```

Most times this characteristic of floating point numbers is not a problem but there are some cases where you need to accurately store large numbers. Examples include tracking a GPS location on the planet's surface or interfacing with digital frequency synthesisers.



As another example, let us say that you want to manipulate time accurately so that you can compare two different date/times to work out which one is earlier. The easy way to do this is to convert the date/time to the number of seconds since some date (say 1st Jan 1900) - then finding the earliest of the two is just a matter of using an arithmetic compare in an IF statement. The problem is that the number of seconds since that date would far exceed the accuracy range of floating point variables and this is where integer variables come in. An integer variable in MMBasic can accurately hold very large numbers to over nine million million million (or  $\pm 9223372036854775807$  to be precise).

The downside of using an integer is that it cannot store fractions (ie, numbers after the decimal point). Any calculation that produces a fractional result will be rounded up or down to the nearest whole number when assigned to an integer.

It is easy to create an integer variable, just add the percent symbol (%) as a suffix to a variable name. For example, sec% is an integer variable. Within a program you can mix integers and floating point and MMBasic will make the necessary conversions but if you want to maintain the full accuracy of integers you should avoid mixing the two.

Just like floating point you can have arrays of integers with up to eight dimensions, all you need to do is add the percent character as a suffix to the array name. For example: days%(365, 5). Beginners often get confused as to when they should use floating point or integers and the answer is simple... always use floating point unless you need a very high level of accuracy in the resulting number. This does not happen often but when you need them you will find that integers are a lifesaver.

## Strings

Strings are another variable type (like floating point and integers). Strings are used to hold a sequence of characters. For example, in the command:

```
PRINT "Hello"
```

The string "Hello" is a string constant. Note that a constant is something that does not change (as against a variable, which can) and that string constants are always surrounded by double quotes. String variables names use the dollar symbol (\$) as a suffix to identify them as a string instead of a normal floating point variable and you can use ordinary assignment to set their value. The following are examples (note that the second example uses an array of strings):

```
Car$ = "Holden"  
Country$(12) = "India"  
Name$ = "Fred"
```

You can also join strings using the plus operator:

```
Word1$ = "Hello"  
Word2$ = "World"  
Greeting$ = Word1$ + " " + Word2$
```

In which case the value of Greeting\$ will be "Hello World".

Strings can also be compared using operators such as = (equals), <> (not equals), < (less than), etc.

For example:

```
IF Car$ = "Holden" THEN PRINT "Was an Aussie made car"
```

The comparison is made using the full ASCII character set so a space will come before a printable character. Also the comparison is case sensitive so 'holden' will not equal "Holden". Using the function UCASE() to convert the string to upper case you can have a case insensitive comparison. For example:

```
IF UCASE$(Car$) = "HOLDEN" THEN PRINT "Was an Aussie made car"
```

You can have arrays of strings but you need to be careful when you declare them as you can rapidly run out of RAM (general memory used for storing variables, etc). This is because MMBasic will by default allocate 255 bytes of RAM for each element of the array. For example, a string array with 100 elements will by default use 25K of RAM. To alleviate this you can use the LENGTH qualifier to limit the maximum size of each element. For instance, if you know that the maximum length of any string that will be stored in the array will be less than 20 characters you can use the following declaration to allocate just 20 bytes for each element:

```
DIM MyArray$(100) LENGTH 20
```

The resultant array will only use 2K of RAM.

## Manipulating Strings

String handling is one of MMBasic's strengths and using a few simple functions you can pull apart and generally manipulate strings.

The basic string functions are:

- |                            |   |
|----------------------------|---|
| LEFT\$(string\$, nbr )     | Returns a substring of <i>string\$</i> with <i>nbr</i> of characters from the left (beginning) of the string.   |
| RIGHT\$(string\$, nbr )    | Same as the above but return <i>nbr</i> of characters from the right (end) of the string.   |
| MID\$(string\$, pos, nbr ) | Returns a substring of <i>string\$</i> with <i>nbr</i> of characters starting from the character <i>pos</i> in the string (ie, the middle of the string). |

For example if S\$ = "This is a string"

Then: R\$ = LEFT\$(S\$, 7)

would result in the value of R\$ being set to: "This is"

and: R\$ = RIGHT\$(S\$, 8)

would result in the value of R\$ being set to: "a string"

finally: R\$ = MID\$(S\$, 6, 2)

would result in the value of R\$ being set to: "is"

Note that in MID\$() the first character position in a string is number 1, the second is number 2 and so on. So, counting the first character as one, the sixth position is the start of the word "is".

Another useful function is:

INSTR(string\$, pattern\$ )	Returns a number representing the position at which <i>pattern\$</i> occurs in <i>string\$</i> .
-----------------------------	--

This can be used to search for a string inside another string. The number returned is the position of the substring inside the main string. Like with MID\$() the start of the string is position 1.

For example if S\$ = "This is a string"

Then: pos = INSTR(S\$, " ")

would result in pos being set to the position of the first space in S\$ (ie, 5).

INSTR() can be combined with other functions so this would return the first **word** in S\$:

```
R$ = LEFT$(S$, INSTR(S$, " ") - 1)
```

There is also an extended version of INSTR():

INSTR(pos, string\$, pattern\$ ) Returns a number representing the position at which *pattern\$* occurs in *string\$* when starting the search at the character position *pos*.

So we can find the second word in S\$ using the following:

```
pos = INSTR(S$, " ")
R$ = LEFT$(S$, INSTR(pos + 1, S$, " ") - 1)
```

This last example is rather complicated so it might be worth working through it in detail so that you can understand how it works.

Note that INSTR() will return the number zero if the sub string is not found and that any string function will throw an error (and halt the program) if that is used as a character position. So, in a practical program you would first check for zero being returned by INSTR() before using that value. The section on serial communications in chapter 7 has a good example of using this function.

## Scientific Notation

Before we finish discussing data types we need to cover off the subject of floating point numbers and scientific notation.

Most numbers can be written normally, for example 11 or 24.5, but very large or small numbers are more difficult. For example, it has been estimated that the number of grains of sand on planet Earth

is 7500000000000000000. The problem with this number is that you can easily lose track of how many zeros there are in the number and consequently it is difficult to compare this with a similar sized number.

A scientist would write this number as  $7.5 \times 10^{18}$  which is called scientific notation and is much easier to comprehend.

MMBasic will automatically shift to scientific notation when dealing with very large or small floating point numbers. For example, if the above number was stored in a floating point variable the PRINT command would display the number as 7.5E+18 (this is BASIC's way of representing  $7.5 \times 10^{18}$ ). As another example, the number 0.0000000456 would display as 4.56E-8 which is the same as  $4.56 \times 10^{-8}$ .

You can also use scientific notation when entering constant numbers in MMBasic. For example:

```
SandGrains = 7.5E+18
```

MMBasic only uses scientific notation for displaying floating point numbers (not integers). For instance, if you assigned the number of grains of sand to an integer variable it would print out as a normal number (with lots of zeros).

## DIM Command

We have used the DIM command before for defining arrays but it can also be used to create ordinary variables. For example, you can simultaneously create four string variables like this:

```
DIM STRING Car, Name, Street, City
```

Note that because these variables have been defined as strings using the DIM command we do not

need the \$ suffix, the definition alone is enough for MMBasic to identify their type. When you use



these variables in an expression you also do not need the type suffix: Eg:

```
City = "Sydney"
```

You can also use the keyword `INTEGER` to define a number of integer variables and `FLOAT` to do the same for floating point variables. This type of notation can also be used to define arrays. For example:

```
DIM INTEGER seconds(200)
```

Another method of defining the variables type is to use the keyword `AS`. For example:

```
DIM Car AS STRING, Name AS STRING, Street AS STRING
```

This is the method used by Microsoft (MMBasic tries to maintain Microsoft compatibility) and it is useful if the variables have different types. For example:

```
DIM Car AS STRING, Age AS INTEGER, Value AS FLOAT
```

You can use any of these methods of defining a variable's type, they all act the same. The advantage of defining variables using the `DIM` command is that they are clearly defined (preferably at the start of the program) and their type (float, integer or string) is not subject to misinterpretation. You can strengthen this by using the following commands at the very top of your program:

```
OPTION EXPLICIT  
OPTION DEFAULT NONE
```

The first specifies to MMBasic that all variables must be explicitly defined using `DIM` before they can be used. The second specifies that the type of all variables must be specified when they are created.

Why are these two commands important?

They can help you to avoid a common programming error which is where you accidentally misspell a variable's name. For example, your program might have the current temperature saved in a variable

called `Temp` but at one point you accidentally misspell it as `Tmp`. This will cause MMBasic to automatically create a variable called `Tmp` and set its value to zero.

This is obviously not what you want and it will introduce a subtle error which could be hard to find – even if you were aware that something was not right. On the other hand, if you used the `OPTION EXPLICIT` command at the start of your program MMBasic would refuse to automatically create the variable and instead would display an error thereby saving you from a probable headache. For small, quick and dirty programs, it is fine to allow MMBasic to automatically create variables but in larger programs you should always disable this feature with `OPTION EXPLICIT` and strengthen it with `OPTION DEFAULT NONE`.

When a variable is created it is set to zero for float and integers and an empty string (ie, contains no

characters) for a string variable. You can set its initial value to something else when it is created using `DIM`. For example:

```
DIM FLOAT nbr = 12.56  
DIM STRING Car = "Ford", City = "Perth"
```

You can also initialise arrays by placing the initialising values inside brackets like this:

```
DIM s$(2) = ("zero", "one", "two")
```

Note that because arrays start from zero this array actually has three elements with the index numbers of 0, 1 and 2. This is why we needed three string constants to initialise it.

## Constants

A common requirement in programming is to define a variable that represents a value without the risk of the value being accidentally changed - which can happen if standard variables were used for this purpose. These are called constants and they can represent pin numbers, signal limits, mathematical constants and so on.

You can create a constant using the CONST command. This defines an identifier that acts like a variable but is set to a value that cannot be changed. For example:

```
CONST BatteryVoltagePin = 26
CONST BatteryMinimum = 11.5
```

These constants can then be used in a program where they make more sense to the casual reader than simple numbers. For example:

```
IF PIN(BatteryVoltagePin) < BatteryMinimum THEN SoundAlarm
```

It is good programming practice to use constants for any fixed number that represents an important value. Normally they are defined at the start of a program where they are easy to see and conveniently located for another programmer to adjust (if necessary). Using the above as an example, you might replace the battery with a different technology and therefore you need to change the minimum battery voltage. That could be easily accomplished if you originally defined this value as a constant at the start of your program.

## Subroutines

A subroutine is a block of programming code which is self contained (like a module) and can be called from anywhere within your program. To your program it looks like a built in MMBasic command and can be used the same. For example, assume that you need a command that would signal an error by printing a message on the console. You could define the subroutine like this:

```
SUB ErrMsg
PRINT "Error detected"
END SUB
```

With this subroutine embedded in your program all you need to do is use the command ErrMsg whenever you want to display the message. For example:

```
IF A < B THEN ErrMsg
```

The definition of a subroutine can be anywhere in the program but typically it is at the end. If MMBasic runs into the definition while running your program it will simply skip over it.

The above example is fine enough but it would be better if a more useful message could be displayed, one that could be customised every time the subroutine was called. This can be done by passing a string to the subroutine as an argument (sometimes called a parameter).

In this case the definition of the subroutine would look like this:

```
SUB ErrMsg Msg$
```

```
PRINT "Error: " + Msg$
END SUB
```

Then when you call the subroutine, you can supply the string to be printed on the command line of the subroutine. For example:

```
ErrMsg "Number too small"
```

When the subroutine is called like this the message "Error: Number too small" will be printed on the console. Inside the subroutine Msg\$ will have the value of "Number too small" and it will be concatenated in the PRINT statement to make the full error message.

A subroutine can have any number of arguments which can be float, integer or string with each argument separated by a comma. Within the subroutine the arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden within the subroutine and be different from arguments defined for the subroutine.

The type of the argument to be supplied can be specified with a type suffix (ie, \$, % or ! for string, integer and float). For example, in the following the first argument must be a string and the second an integer:

```
SUB MySub Msg$, Nbr%
...
END SUB
```

MMBasic will convert the supplied values if it can, so if your program supplied a floating point value as the second argument MMBasic will convert it to an integer. If MMBasic cannot convert the value it will display an error. For example, if you supplied a string for the second argument your program will stop with an error.

You do not have to use the type suffixes, you can instead define the type of the arguments using the

AS keyword similar to the way it is used in the DIM command. For example, the following is identical to the above example:

```
SUB MySub Msg AS STRING, Nbr AS INTEGER
...
END SUB
```

Of course, if you used only one variable type throughout the program and used OPTION DEFAULT to set that type you could ignore the question of variable types completely.

When a subroutine is called with an argument that is a variable (ie, not a constant or expression) MMBasic will create a corresponding variable within the subroutine *that points back to this variable*. Any changes to the variable representing the argument inside the subroutine will also change the variable used in the call. This is called passing arguments by reference.

This is best explained by example:

```
DIM MyNumber = 5           ' set the variable to 5
CalcSquare MyNumber        ' the subroutine will square its value
PRINT MyNumber             ' this will print the number 25
END                        ' square the argument and pass it back
```

```
SUB CalcSquare nbr
    nbr = nbr * nbr
END SUB
```

The subroutine CalcSquare will take its argument, square it and write it back to the variable representing the argument (nbr). Because the subroutine was called with a variable (MyNumber) the variable nbr will point back to MyNumber and any change to nbr will also change MyNumber accordingly. As a result the PRINT statement will output 25.

Passing arguments by reference is handy because it allows a subroutine to pass back values to the

code that called it. However it can lead to trouble if a subroutine used the variable representing an argument as a general purpose variable and changed its value. Then, if it were called with a variable

as an argument, that variable would be inadvertently changed. For this reason you should avoid manipulating variables representing arguments inside a subroutine, instead assign the value to a local variable (see below) and manipulate that instead.

When you call a subroutine you can omit some (or all) of the parameters and they will take the value of zero (for floats or integers) or an empty string. This is handy as your subroutine can tell if a parameter is missing and act accordingly.

For example, here is our subroutine to generate an error message but this version can be used without specifying an error message as a parameter:

```
SUB ErrMsg Msg$
    IF Msg$ = "" THEN
        PRINT "Error detected"
    ELSE
        PRINT "Error: " + Msg$
    ENDIF
END SUB
```

Within a subroutine you can use most features of MMBasic including calling other subroutines, IF...THEN commands, FOR...NEXT loops and so on. However one thing that you cannot do is jump out of a subroutine using GOTO (if you do the result will be undefined). Normally the subroutine will exit when the END SUB command is reached but you can also terminate the subroutine early by using the EXIT SUB command.

## Functions

Functions are similar to subroutines with the main difference being that a function is used to return a value in an expression. For example, if you wanted a function to convert a temperature from degrees Celsius to Fahrenheit you could define:

```
FUNCTION Fahrenheit(C)
    Fahrenheit = C * 1.8 + 32
END FUNCTION
```

Then you could use it in an expression:

```
Input "Enter a temperature in Celsius: ", t
PRINT "That is the same as" Fahrenheit(t) "F"
```

Or as another example:

```
IF Fahrenheit(temp) <= 32 THEN PRINT "Freezing"
```

You could also define the reverse:

```

FUNCTION Celsius(F)
    Celsius = (F - 32) * 0.5556
END FUNCTION

```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value is made available to the expression that called it. The rules for the argument list in a function are similar to subroutines. The only difference is that parentheses are always required around the argument list when you are calling a function, even if there are no arguments (they are optional when calling a subroutine). Functions can also use LOCAL and STATIC variables just like subroutines.

To return a value from the function you assign a value to the function's name within the function. If the function's name is terminated with a type suffix (ie, \$, a % or a !) the function will return that type (string, integer or float), otherwise it will return whatever the OPTION DEFAULT is set to. For example, the following function will return a string:

```

FUNCTION LVal$(nbr)
    IF nbr = 0 THEN LVal$ = "False" ELSE LVal$ = "True"
END FUNCTION

```

You can explicitly specify the type of the function by using the AS keyword and then you do not need to use a type suffix (similar to defining a variable using DIM). This is an example:

```

FUNCTION LVal(nbr) AS STRING
    IF nbr = 0 THEN LVal = "False" ELSE LVal = "True"
END FUNCTION

```

In this case the type returned by the function LVal will be a string.

As for subroutines you can use most features of MMBasic within functions. This includes FOR...NEXT loops, calling other functions and subroutines, etc. Also, the function will return to the expression that called it when the END FUNCTION command is reached but you can also return early by using the EXIT FUNCTION command.

## Local Variables

Variables that are created using DIM or that are just automatically created are called *global* variables. This means that they can be seen and used anywhere in the program including within subroutines. However, inside a subroutine you will often need to use variables for various tasks that

are internal to the subroutine. In portable code you do not want the name you chose for such a variable to clash with a global variable of the same name. To this end you can define a variable using the LOCAL command within the subroutine.

The syntax for LOCAL is identical to the DIM command, this means that the variable can be an array, you can set the type of the variable and you can initialise it to some value.

For example, this is our ErrMsg subroutine that we used a few pages back but this time it has been extended to use a local variable for joining the error message strings.

```

SUB ErrMsg Msg$
    LOCAL STRING tstr
    tstr = "Error: " + Msg$
    PRINT tstr
END SUB

```



The variable `tstr` is declared as `LOCAL` within the subroutine, which means that (like the argument list) it will only exist within the subroutine and will vanish when the subroutine exits. You can have a global variable called `tstr` in your main program and it will be different from the variable `tstr` in the subroutine (in this case the global `tstr` will be hidden within the subroutine). You should always use local variables for operations within your subroutine because they help make the subroutine much more self contained and portable.

## Static Variables

`LOCAL` variables are reset to their initial values (normally zero or an empty string) every time the subroutine starts, however there are times when you would like the variable to retain its value between calls to the subroutine. This type of variable is defined with the `STATIC` command. We can demonstrate how `STATIC` variables are useful by extending the `ErrMsg` subroutine to prevent duplicated calls to the subroutine repeatedly displaying the same message. For example, our program might call this subroutine from multiple places but if the message is the same in a number of subsequent calls we would like to see the message just once. To keep track of the last message displayed we use a static variable called `lastmsg`. This will hold the text of the last message and we can compare it to the current message text to determine if it is different and therefore should be printed. This is our new subroutine:

```
SUB ErrMsg Msg$
    STATIC STRING lastmsg
    LOCAL STRING tstr
    IF Msg$ <> lastmsg THEN
        tstr = "Error: " + Msg$
        PRINT tstr
        lastmsg = Msg$
    ENDIF
END SUB
```

This would give just one message every time a call is made with a duplicate message text. The `STATIC` command uses exactly the same syntax as `DIM`. This means that you can define different types of static variables including arrays and you can also initialise them to some value. The static variable is created on the first time the `STATIC` command is encountered and it is automatically set to zero (if a float or integer) or an empty string. On subsequent calls to the subroutine `MMBasic` will recognise that the variable has already been created and it will leave its value untouched (ie, whatever it was in the previous call). As with `DIM` you can also initialise a static variable to some value. For example:

```
STATIC INTEGER var = 123
```

On the first call (when the variable is created) it will be initialised to 123 but on subsequent calls it will keep whatever its value was previously set to. Mostly static variables are used to keep track of the *state* of something while inside a subroutine. A *state* is a record of something that has happened previously. Examples include:

- Has the COM port already been opened?
- What steps in a sequence have we completed?
- What text has already been displayed?

Normally you will use global variables (created using `DIM`) to track a *state* but sometimes you want

this to be contained within a subroutine and this is where static variables are valuable. Just like LOCAL the use of STATIC helps to make your subroutines more self contained and portable.

## Calculate Days

We have covered a lot of programming commands and techniques so far in this tutorial and, to give

an example of how they work together, the following is an example program that will calculate the number of days between two dates.

It works by getting two dates from the user at the console and then converting them to integers representing the number of days since 1900. With these two numbers a simple subtraction will give the number of days between them.

' Example program to calculate the number of days between two dates

```
OPTION EXPLICIT
OPTION DEFAULT NONE
DIM STRING s
DIM FLOAT d1, d2
DO
    PRINT
    PRINT "Enter the date as dd mmm yyyy"
    PRINT " First date";
    INPUT s
    d1 = GetDays(s)
    IF d1 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
    PRINT "Second date";
    INPUT s
    d2 = GetDays(s)
    IF d2 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
    PRINT "Difference is" ABS(d2 - d1) " days"
LOOP

' Calculate the number of days since 1/1/1900
FUNCTION GetDays(d$) AS FLOAT
    LOCAL STRING Month(11) =
        ("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
    LOCAL FLOAT Days(11) = (0,31,59,90,120,151,181,212,243,273,304,334)
    LOCAL FLOAT day, mth, yr, s1, s2

    ' Find the separating space character within a date
    s1 = INSTR(d$, " ")
    IF s1 = 0 THEN EXIT FUNCTION
    s2 = INSTR(s1 + 1, d$, " ")
    IF s2 = 0 THEN EXIT FUNCTION

    ' Get the day, month and year as numbers
    day = VAL(MID$(d$, 1, s2 - 1)) - 1
    IF day < 0 OR day > 30 THEN EXIT FUNCTION
    FOR mth = 0 TO 11
        IF LCASE$(MID$(d$, s1 + 1, 3)) = Month(mth) THEN EXIT FOR
    NEXT mth
    IF mth > 11 THEN EXIT FUNCTION
```

```
yr = VAL(MID$(d$, s2 + 1)) - 1900  
IF yr < 1 OR yr >= 200 THEN EXIT FUNCTION
```

```
' Calculate the number of days including adjustment for leap years  
GetDays = (yr * 365) + FIX((yr - 1) / 4)  
IF yr MOD 4 = 0 AND mth >= 2 THEN GetDays = GetDays + 1  
GetDays = GetDays + Days(mth) + day
```

```
END FUNCTION
```

Note that the line starting LOCAL STRING Month(11) has been wrapped around because of the limited page width – it is one line as follows:

```
LOCAL STRING Month(11) = ("jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec")
```

When this program is run it will ask for the two dates to be entered and you need to use the form of dd mmm yyyy. This screen capture shows what the running program will look like.

The main feature of the program is the user defined function

GetDays() which takes a string entered at the console, splits it into its day, month and year components then calculates the number of days since 1st January 1900. This function is called twice, once for the first date and then again for the second date. It is

then just a matter of subtracting one date (in days) from the other to get the difference in days.

We will not go into the detail of how the calculations are made (ie, handling leap years) as that can be left as an exercise for the reader. However it is appropriate to point out some features of MMBasic that are used by the program.

It demonstrates how local variables can be used and how they can be initialised. In the function GetDays() two arrays are declared and initialised at the same time. These are just a convenient method of looking up the names of the months and the cumulative number of days for each month. Later in the function (the FOR loop) you can see how they make dealing with twelve different months quite efficient.

Another feature highlighted by this program is the string handling features of MMBasic. The INSTR() function is used to locate the two space characters in the date string and then later the MID\$() function uses these to extract the day, month and year components of the date. The VAL() function is used to turn a string of digits (like the year) into a number that can be stored in a numeric variable.

Note that the value of a function is initialised to zero every time the function is run and unless it is set to some value it will return a zero value. This makes error handling easy because we can just exit the function if an error is discovered. It is then the responsibility of the calling program code to check for a return value of zero which signifies an error.

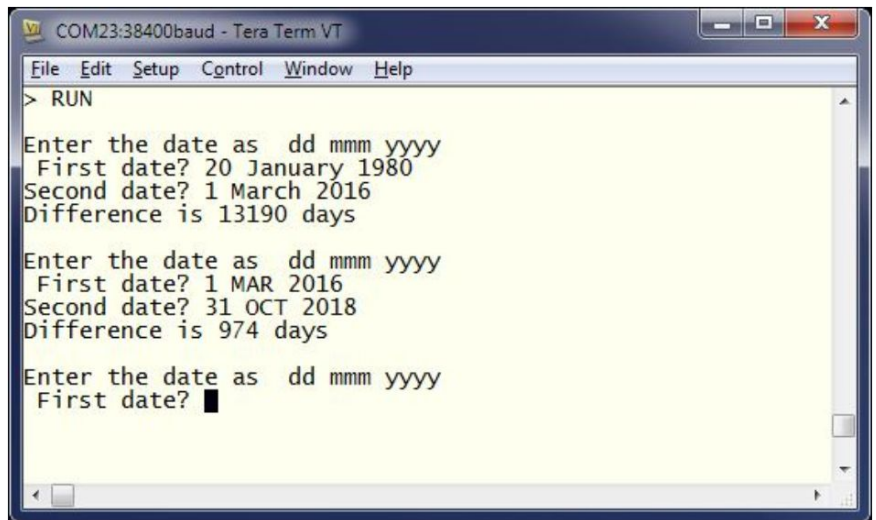
This program illustrates one of the benefits of using subroutines and functions which is that when written and fully tested they can be treated as a trusted "black box" that does not have to be opened.

For this reason functions like this should be properly tested and then, if possible, left untouched (in case you add some error).

There are a few features of this program that we have not covered before. The first is the MOD operator which will calculate the remainder of dividing one number into another. For example, if you divided 4 into 15 you would have a remainder of 3 which is exactly what the expression 15 MOD 4 will return. The ABS() function is also new and will return its argument as a positive number (eg, ABS(-15) will return +15 as will ABS(15)).

The EXIT FOR command will exit a FOR loop even though it has not reached the end of its looping, EXIT FUNCTION will immediately exit a function even though execution has not reached the end of the function and CONTINUE DO will immediately cause the program to jump to the end of a DO loop and execute it again.

Why would this program be useful? Well some people like to count their age in days, that way every day is a birthday! You can calculate your age in days, just enter the date that you were born and today's date. That is not particularly useful but the program itself is valuable as it demonstrates many of the characteristics of programming in MMBasic. So, pull out the ft-RPI-sa *User Manual* and work your way through the program code – it should be a rewarding experience.



```
COM23:38400baud - Tera Term VT
File Edit Setup Control Window Help
> RUN

Enter the date as dd mmm yyyy
First date? 20 January 1980
Second date? 1 March 2016
Difference is 13190 days

Enter the date as dd mmm yyyy
First date? 1 MAR 2016
Second date? 31 OCT 2018
Difference is 974 days

Enter the date as dd mmm yyyy
First date? █
```

# Good Programming Habits

Before we finish with the subject of BASIC programming it will be worth while providing some hints on how to write programs that are easy to understand and maintain. This can be more important than one might think. A poorly written program is more likely to contain bugs and people will be reluctant to try and fix such a program because the logic is hard to understand. You might think that this does not matter because you will be the only person to ever read the program. But your memory will fade over time and when you try to modify a program that you wrote (say) a year ago it will be the same as if it was written by a complete stranger who you will hope had followed these hints.

1. Use lots of comments. They are the first thing that people (including you) will read when they pick up your program and they are invaluable in rendering the program and its logic understandable. Even if no one else will read the program you will appreciate the comments in the future.
2. Use indenting to illustrate the logic of loops, multi line IF statements, subroutines, etc. Without indenting the casual reader would have to search many lines to determine when a block of code has terminated.
3. Keep the comments and indenting up to date. When modifying a program it is easy to forget that these features also need updating and nothing is worse than a misleading comment or indentation.
4. Define all variables using DIM or LOCAL statements at the start of the program, subroutine or function. Do not let variables be automatically created, instead use OPTION EXPLICIT and OPTION DEFAULT NONE.
5. Use variable names that make sense. For a simple loop you can use a short variable like 'spd' but for something important that is scattered throughout the program use a descriptive variable name such as 'MaxSpeedLimit'.
6. Define significant numbers as constants at the start of the program using the CONST command.
7. MMBasic does not worry about upper or lower case characters in identifiers (variable names, subroutine names, etc) but regardless, you should use a consistent case. For example, you can use either MaxSpeedLimit or maxspeedlimit in a program, but you should not use both.
8. Package unique and self contained pieces of code into a subroutine or function. These have limited entry/exit points which means that someone can read through such a module and more easily satisfy themselves that it is working correctly. From then on it can be treated as a trusted portion of code.
9. Don't use the GOTO command unless you absolutely have to. Features such as multiline IF statements, subroutines and functions are much easier to understand than a program which uses GOTOs to jump around.
10. Don't be obsessed with optimising your code to make it faster for MMBasic to interpret. MMBasic makes many optimisations of its own and anything that you do will have little effect on speed and may obscure the logic of the program. Normally only a very small part of a program needs to run fast and if that is the case that portion would be better being written in C which can be embedded in the BASIC program.

For a short program you can ignore many of these hints but for a large program they can be a huge help and may help prevent your hair turning prematurely grey if you need to modify your program in the future.

# Commands

Note that the commands related to the I<sup>2</sup>C and Output configuration are not listed here but are described in the appendices at the end of this document.

Square brackets indicate that the parameter or characters are optional.

ft-RPI-sa specific commands are colored in orange.

' (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
? (question mark)	Shortcut for the PRINT command.
AUTOSAVE or AUTOSAVE CRUNCH	Enter automatic program entry mode. This command will take lines of text from the console serial input and save them to memory. This mode is terminated by entering Control-Z which will then cause the received data to be saved into program memory overwriting the previous program. The CRUNCH option instructs MMBasic to remove all comments, blank lines and unnecessary spaces from the program before saving. This can be used on large programs to allow them to fit into limited memory. CRUNCH can be abbreviated to the single letter C. At any time this command can be aborted by Control-C which will leave program memory untouched. This is one way of transferring a BASIC program into the S32K1xx. The program to be transferred can be pasted into a terminal emulator and this command will capture the text stream and store it into program memory. It can also be used for entering a small program directly at the console input.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
CONST id = expression [, id = expression] ... etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or \$) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.



DATA constant[,constant]...	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.
DATE\$ = "DD-MM-YY" or DATE\$ = "DD/MM/YY"	Set the date of the internal clock/calendar. DD, MM and YY are numbers, for example: DATE\$ = "28-7-2014" The date is set to "01-01-2000" on power up.
DIM [type] decl [,decl]... where 'decl' is: var [length] [type] [init] 'var' is a variable name with optional dimensions 'length' is used to set the maximum size of the string to 'n' as in LENGTH n 'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT) 'init' is the value to initialise the variable and consists of: = <expression> For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used. Examples: DIM nbr(50) DIM INTEGER nbr(50) DIM name AS STRING DIM a, b\$, nbr(100), strn\$(20) DIM a(5,5,5), b(1000) DIM strn\$(200) LENGTH 20 DIM STRING strn(200) LENGTH 20 DIM a = 1234, b = 345 DIM STRING strn = "text" DIM x%(3) = (11, 22, 33, 44)	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter). When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced. The type of the variable (ie, string, float or integer) can be specified in one of three ways: By using a type suffix (ie, !, % or \$ for float, integer or string). For example: DIM nbr%, amount!, name\$ By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example: DIM STRING first_name, last_name, city By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable. For example: DIM amount AS FLOAT, name AS STRING Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example: DIM STRING city = "Perth", house = "Brick" The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax. As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with a number of dimensions). Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example: DIM array(10, 20) Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1. The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and as each floating point number

	<p>on the S32K1xx requires 4 bytes, a total of 924 bytes of memory will be allocated.</p> <p>Strings will default to allocating 255 bytes (ie, characters) of memory for each element and this can quickly use up memory when defining arrays of strings. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.</p> <p>For example: DIM STRING s(5, 10) will declare a string array with 66 elements consuming 16,896 bytes of memory while:</p> <pre>DIM STRING s(5, 10) LENGTH 20</pre> <p>Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.</p> <p>If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays. This keyword can also be used with non array string variables but it will not save any memory.</p> <p>In the above example you can also use the Microsoft syntax of specifying the type after the length qualifier. For example:</p> <pre>DIM s(5, 10) LENGTH 20 AS STRING</pre> <p>Arrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example:</p> <pre>DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)</pre> <p>or</p> <pre>DIM s\$(3) = ("foo", "boo", "doo", "zoo")</pre> <p>Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc.</p> <p>Also note that the initialising values must be after the LENGTH qualifier (if used) and after the type declaration (if used).</p>
ENDIF or END IF	<p>Terminates a multiline IF statement.</p> <p>See the multiline IF statement for more details.</p>
END SUB	<p>Marks the end of a user defined subroutine. See the SUB command.</p> <p>Each sub must have one and only one matching END SUB statement.</p> <p>Use EXIT SUB if you need to return from a subroutine from within its body.</p>

EPULL <input>,<pull config>	<p>Configures the input pull resistor on each Ix input. Each bit in the value given with the &lt;input&gt; parameter represents the appropriate input (e.g. Bit 0 = I1, Bit 1 = I2 etc.) Inputs which are set to „0“ are configured Hi-Z i.e. floating. The &lt;pull config&gt; parameter selects the pull-resistor polarity for each given input. Setting the appropriate bit to „0“ configures the pull to be a pull-down and a „1“ configures the pull to be a pull-up resistor.</p> <p>Examples:</p> <p>Epull &amp;H01,&amp;H00 Configures pull resistor on input I1 to a pull-down.</p> <p>Epull &amp;H0F,&amp;H0A Configures pull resistor on inputs I4 &amp; I2 to pull-up and input I3 and I1 to pull-down.</p> <p>Epull &amp;HFF,&amp;HF0 Configures pull resistor on inputs I8-I5 to pull-up and input I4-I1 to pull down.</p> <p>The pull-up voltage is set to 5V and the resistor used for any pull is set to 2k7=2.7kΩ=2700Ohm</p>
ERASE variable [,variable]...	<p>Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, dat()) or just by specifying the variable's name (eg, dat).</p> <p>Use CLEAR to delete all variables at the same time (including arrays).</p>
ERROR [error_msg\$]	<p>Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.</p>
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	<p>EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.</p>
FOR counter = start TO finish [STEP increment]	<p>Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' is greater than 'finish'.</p> <p>The 'increment' can be an integer or floating point number. Note that using a floating point fractional number for 'increment' can accumulate rounding errors in 'counter' which could cause the loop to terminate early or late.</p> <p>'increment' can be negative in which case 'finish' should be less than 'start' and the loop will count downwards.</p> <p>See also the NEXT command.</p>
FUNCTION xxx (arg1 [,arg2, ...]) [AS <type>} <statements> <statements> xxx = <return value>	<p>Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program.</p> <p>'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx\$) or by specifying the type using AS &lt;type&gt; at the end of the</p>

END FUNCTION	<p>functions definition. For example:  FUNCTION xxx (arg1, arg2) AS STRING  'arg1', 'arg2', etc are the arguments or parameters to the function (the brackets are always required, even if there are no arguments). An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1\$) or by specifying the type using AS &lt;type&gt; (ie, arg1 AS STRING).  The argument can also be another defined function or the same function if recursion is to be used (the recursion stack is limited to 50 nested calls).  To set the return value of the function you assign the value to the function's name. For example:  FUNCTION SQUARE(a)  SQUARE = a * a  END FUNCTION  Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit.  You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example:  PRINT SQUARE(56.8)  When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.  Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string. Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable and therefore may be accessed after the function has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference and must be the correct type.  You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including the possibility of ruining your day.</p>
GOTO target	<p>Branches program execution to the target, which can be a line number or a label.</p>
IF expr THEN stmt [: stmt] or IF expr THEN stmt ELSE stmt	<p>Evaluates the expression 'expr' and performs the statement following the THEN keyword if it is true or skips to the next line if false. If there are more statements on the line (separated by colons (:)) they will also be executed if true or skipped if false. The ELSE keyword is optional and if present the statement(s) following it will be executed if 'expr' resolved to be false.  The 'THEN statement' construct can be also replaced with:  GOTO linenummer   label'.  This type of IF statement is all on one line.</p>

IF expression THEN <statements> [ELSEIF expression THEN <statements>] [ELSE <statements>] ENDIF	Multiline IF statement with optional ELSE and ELSEIF cases and ending with ENDIF. Each component is on a separate line. Evaluates 'expression' and performs the statement(s) following THEN if the expression is true or optionally the statement(s) following the ELSE statement if false. The ELSEIF statement (if present) is executed if the previous condition is false and it starts a new IF chain with further ELSE and/or ELSEIF statements as required. One ENDIF is used to terminate the multiline IF.
INPUT ["prompt\$";] var1 [,var2 [, var3 [, etc]]]	Will take a list of values separated by commas (,) entered at the console and will assign them to a sequential list of variables. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 The list of variables can be a mix of float, integer or string variables. The values entered at the console must correspond to the type of variable. If a single value is entered a comma is not required (however that value cannot contain a comma). 'prompt\$' is a string constant (not a variable or expression) and if specified it will be printed first. Normally the prompt is terminated with a semicolon (;) and in that case a question mark will be printed following the prompt. If the prompt is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command. For example: Var = 56
LIBRARY SAVE or LIBRARY DELETE or LIBRARY LIST	The library is a special segment of program memory that can contain program code such as subroutines and functions. These routines are not visible to the programmer but are available to any program running on the S32K1xx and act the same as built in commands and functions in MMBasic. See the chapter "Special Functions and the Library" earlier in this manual for a full explanation. LIBRARY SAVE will take whatever is in normal program memory, compress it (remove redundant data such as comments) and append it to the library area (main program memory is then empty). The code in the library will not show in LIST or EDIT and will not be deleted when a new program is loaded or NEW is used. LIBRARY DELETE will remove the library and recover the memory used. LIBRARY LIST will list the contents of the library. Note that any code in the library that is not contained within a subroutine or function will be executed immediately before a program is run. This can be used to initialize constants, set options, etc.
LINE INPUT [prompt\$,] string-variable\$	Reads an entire line from the console input into 'string-variable\$'. 'prompt\$' is a string constant (not a variable or expression) and if specified it will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of 'prompt\$'.

LIST or LIST ALL	List a program on the serial console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses. This is useful if you wish to transfer the program in the S32K1xx to a terminal emulator on a PC that has the ability to capture its input stream to a file.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
MEMORY	List the amount of memory currently in use. For example: Flash: 21K (35%) Program (805 lines) 1K ( 1%) 4 Saved Variables 37K (63%) Free RAM: 9K (16%) 5 Variables 18K (32%) General 26K (52%) Free Notes: <ul style="list-style-type: none"> <li>🐾 Memory usage is rounded to the nearest 1K byte.</li> <li>🐾 Program memory is cleared by the NEW command.</li> <li>🐾 General memory is used by serial I/O buffers, etc.</li> <li>🐾 Variables and the general memory spaces are cleared by many commands (eg, NEW, RUN, etc) as well as the specific commands CLEAR and ERASE.</li> <li>🐾 When a program is loaded it is first buffered in RAM which limits the maximum program size. MMBasic tokenises the program when it is stored in flash so the final size in flash might vary from this.</li> </ul>
NEW	Deletes the program in flash and clears all variables including saved variables.
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple counter-variables as in: NEXT x, y, z



ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	<p>This controls the action taken if an error occurs while running a program and applies to all errors discovered by MMBasic including syntax errors, wrong data, missing hardware, etc.</p> <p>ON ERROR ABORT will cause MMBasic to display an error message, abort the program and return to the command prompt. This is the normal behaviour and is the default when a program starts running.</p> <p>ON ERROR IGNORE will cause any error to be ignored.</p> <p>ON ERROR SKIP will ignore an error in a number of commands (specified by the number 'nn') executed following this command. 'nn' is optional, the default if not specified is one. After the number of commands has completed (with an error or not) the behaviour of MMBasic will revert to ON ERROR ABORT.</p> <p>If an error occurs and is ignored/skipped the read only variable MM.ERRNO will be set to non zero and MM.ERRMSG\$ will be set to the error message that would normally be generated. These are reset to zero and an empty string by ON ERROR CLEAR. They are also cleared when the program is run and when ON ERROR IGNORE and ON ERROR SKIP are used.</p> <p>ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used</p>
OPTION AUTORUN OFF   ON	<p>Instruct MMBasic to automatically run the program stored in flash when it starts up or is restarted by the WATCHDOG command. This is turned off by the NEW command but other commands that might change program memory (EDIT, etc) do not change this setting.</p> <p>Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt despite this option.</p>
OPTION BASE 0   1	<p>Set the lowest value for array subscripts to either 0 or 1.</p> <p>This must be used before any arrays are declared and is reset to the default of 0 on power up.</p>
OPTION DEFAULT FLOAT   INTEGER   STRING   NONE	<p>Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined.</p> <p>When a program is run the default is set to FLOAT for compatibility with previous versions of MMBasic.</p>
OPTION EXPLICIT	<p>Placing this command at the start of a program will require that every variable be explicitly declared using the DIM command before it can be used in the program.</p> <p>This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.</p>
OPTION LIST	<p>This will list the settings of any options that have been changed from their default setting and are the type that is saved in flash.</p>
OPTION RESET	<p>Reset all saved options (including the PIN) to the default values.</p>
OPTION SAVE	<p>Used to save configuration options that are embedded in a program. See the section "Special Functions and the Library" for more details.</p>
OPTION TAB 2   4   8	<p>Set the spacing for the tab key. Default is 2.</p> <p>This option will be remembered even when the power is removed.</p>
OUTPUT <mode> <parameter>	<p>Please refer to appendix B for more details on this command.</p>

PAUSE delay	<p>Halt execution of the running program for 'delay' ms. This can be a fraction. For example, 0.2 is equal to 200 <math>\mu</math>s. The maximum delay is 2147483647 ms (about 24 days).</p> <p>Note that interrupts will be recognised and processed during a pause.</p>
POKE BYTE addr%, byte or POKE WORD addr%, word% or POKE INTEGER addr%, int% or POKE FLOAT addr%, float! or POKE VAR var, offset, byte or POKE VARTBL, offset, byte	<p>Will set a byte or a word within the S32K1xx virtual memory space.</p> <p>POKE BYTE will set the byte (ie, 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer.</p> <p>POKE WORD will set the word (ie, 32 bits) at the memory location 'addr%' to 'word%'. 'addr%' and 'word%' should be integers.</p> <p>POKE INTEGER will set the MMBasic integer (ie, 64 bits) at the memory location 'addr%' to int%. 'addr%' and int%' should be integers.</p> <p>POKE FLOAT will set the word (ie, 32 bits) at the memory location 'addr%' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number.</p> <p>POKE VAR will set a byte in the memory address of 'var'. 'offset' is the <math>\pm</math>offset from the address of the variable. An array is specified as var().</p> <p>POKE VARTBL will set a byte in MMBasic's variable table. 'offset' is the <math>\pm</math>offset from the start of the variable table. Note that a comma is required after the keyword VARTBL.</p> <p>For backwards compatibility the old form of POKE hiword, loword, val is still accepted. In this case the address is specified by 'hiword' which is the top 16 bits of the address while 'loword' is the bottom 16 bits.</p> <p>This command is for expert users only. The S32K1xx maps all control registers, flash (program) memory and volatile (RAM) memory into a single address space. The S32K1 Reference Manual lists all the details of this address space.</p>
PRINT expression [[,; ]expression] ... etc	<p>Outputs text to the serial console followed by a carriage return/newline pair.</p> <p>Multiple expressions can be used and must be separated by either a:</p> <ul style="list-style-type: none"> <li>➤ Comma (,) which will output the tab character</li> <li>➤ Semicolon (;) which will not output anything (it is just used to separate expressions).</li> <li>➤ Nothing or a space which will act the same as a semicolon.</li> </ul> <p>A semicolon (;) or a comma (,) at the end of the expression list will suppress the output of the carriage return/newline pair at the end of a print statement.</p> <p>When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large or small floating point numbers are automatically printed in scientific number format.</p> <p>The function TAB() can be used to space to a certain column and the STR\$( ) function can be used to justify or otherwise format strings.</p>
RANDOMIZE nbr	<p>Seed the random number generator with 'nbr'.</p> <p>On power up the random number generator is seeded with zero and will generate the same sequence of random numbers each time. To generate a different random sequence each time you must use a different value for 'nbr' (the TIMER function is handy for that).</p>

READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. See also DATA and RESTORE.										
REM string	REM allows remarks to be included in a program. Note the Microsoft style use of the single quotation mark (') to denote remarks is also supported and is preferred.										
RESET	Assert a reset on the MCU and restart/reinitialize the board. If the option „OPTION AUTORUN ON“ is set, the stored program will start immediately after re-initialization.										
RESTORE [line]	Resets the line and position counters for the READ statement. If 'line' is specified the counters will be reset to the beginning of the specified line. 'line' can be a line number or label. If 'line' is not specified the counters will be reset to the start of the program.										
RTC GETTIME	RTC GETTIME will get the current date/time from the NXP PCF2127 real time clock and set the internal MMBasic clock accordingly. The date/time can then be retrieved with the DATE\$ and TIME\$ functions.										
RTC SETTIME year, month, day, hour, minute, second	RTC SETTIME will set the time in the clock chip. 'hour' must use 24 hour notation.										
RTC SETALARM day, hour, minute, second, enable	<p>RTC SETALARM will set the alarm time in the clock chip and once the alarm happens, the Raspberry Pi may switch on, depending on J4 setting. 'hour' must use 24 hour notation. It's not possible to use year and month for alarm setting.</p> <p>&lt;enable&gt; specifies which setting is taken into account for the alarming condition and ranges from 1 to 15.</p> <table border="1"> <thead> <tr> <th>Bit #</th><th>If Bit set, following is taken into account</th></tr> </thead> <tbody> <tr> <td>0 (=1 decimal)</td><td>second</td></tr> <tr> <td>1 (=2 decimal)</td><td>minute</td></tr> <tr> <td>2 (=4 decimal)</td><td>hour</td></tr> <tr> <td>3 (=8 decimal)</td><td>day</td></tr> </tbody> </table> <p>e.g.:            RTC SETALARM 1,12,0,0,15 will create an alarm every 1<sup>st</sup> of the month at 12:00:00            RTC SETALARM 1,6,30,0,6 will create an alarm which occurs every day at 6:30 (the 1 for the day is used as a dummy as it has to be present to keep the syntax)            RTC SETALARM 1,1,30,27,3 will create an alarm at 30m27s every hour and every day.</p> <p>An alarm event can be monitored by the system variable RTCALARM.</p>	Bit #	If Bit set, following is taken into account	0 (=1 decimal)	second	1 (=2 decimal)	minute	2 (=4 decimal)	hour	3 (=8 decimal)	day
Bit #	If Bit set, following is taken into account										
0 (=1 decimal)	second										
1 (=2 decimal)	minute										
2 (=4 decimal)	hour										
3 (=8 decimal)	day										
RTC SETALARM OFF	RTC SETALARM OFF disables all alarm settings.										

RTC CLEARALARM	Once an alarm occurred, it need to be cleared in the RTC as otherwise the supply for the Raspberry Pi can't be switched off completely anymore.
RTC GETRAM addr, var RTC SETRAM addr, var	Used to read and write to the battery backed RAM of the RTC device. 'addr' reflects the address the data is written to or read from and can any value between 0 and 511. 'var' contains the value to be written but can be also a variable containing the write data or to store the read data.
RTC SETREG reg, value RTC GETREG reg, var	<p>The RTC SETREG and GETREG commands can be used to set or read the contents of registers within the RTC chip. 'reg' is the register's number, 'value' is the number to store in the register and 'var' is a variable that will receive the number read from the register. These commands are not necessary for normal operation but they can be used to manipulate special features of the chip (alarms, output signals, etc). They are also useful for storing temporary information in the chip's battery backed RAM.</p> <p>This chip is an I<sup>2</sup>C device and is connected to the two I<sup>2</sup>C pins with appropriate pull-up resistors. If the I<sup>2</sup>C bus is already open the RTC command will use the current settings, otherwise it will temporarily open the connection with a speed of 100 kHz.</p>
RUN	Run the program held in flash memory.

<pre> SELECT CASE value CASE testexp [[, testexp] ...] &lt;statements&gt; &lt;statements&gt; CASE ELSE &lt;statements&gt; &lt;statements&gt; END SELECT </pre>	<p>Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression.</p> <p>'testexp' is the value that is to be compared against. It can be:</p> <ul style="list-style-type: none"> <li>➤ A single expression (i.e. 34, "string" or E_Dig(4)*5) to which it may equal</li> <li>➤ A range of values in the form of two single expressions separated by the keyword "TO" (ie, 5 TO 9 or "aa" TO "cc")</li> <li>➤ A comparison starting with the keyword "IS" (which is optional). For example: IS &gt; 5, IS &lt;= 10.</li> </ul> <p>When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true.</p> <p>If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any &lt;statements&gt; and continue with the code following the END SELECT.</p> <p>When a match is made the &lt;statements&gt; following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT.</p> <p>An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT.</p> <p>Example:</p> <pre> SELECT CASE nbr% CASE 4, 9, 22, 33 TO 88 statements CASE IS &lt; 4, IS &gt; 88, 5 TO 8 statements CASE ELSE statements END SELECT </pre> <p>Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT...CASE statements can be nested inside the CASE statements of other SELECT...CASE statements</p>
<pre> STATIC variable [, variables] See DIM for the full syntax. </pre>	<p>Defines a list of variable names which are local to the subroutine or function.</p> <p>These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command).</p> <p>This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters.</p> <p>Static variables can be initialized to a value. This initialization will take effect only on the first call to the subroutine (not on subsequent calls).</p>

SUB xxx (arg1 [,arg2, ...]) <statements> <statements> END SUB	<p>Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program.</p> <p>'xxx' is the subroutine name and it must meet the specifications for naming a variable.</p> <p>'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1\$) or by specifying the type using AS &lt;type&gt; (ie, arg1 AS STRING).</p> <p>Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine.</p> <p>The command EXIT SUB can be used for an early exit.</p> <p>You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: MySub a1, a2</p> <p>When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string.</p> <p>Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.</p>
TIME\$ = "HH:MM:SS" or TIME\$ = "HH:MM" or TIME\$ = "HH"	<p>Sets the time of the internal clock. MM and SS are optional and will default to zero if not specified. For example TIME\$ = "14:30" will set the clock to 14:30 with zero seconds.</p> <p>The time is set to "00:00:00" on power up.</p>
TIMER = msec	<p>Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer.</p> <p>See the TIMER function for more details.</p>
TRACE ON or TRACE OFF	<p>TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs.</p>



XMODEM SEND or XMODEM RECEIVE or XMODEM CRUNCH	<p>Transfers a BASIC program to or from a remote computer using the XModem protocol. The transfer is done over the serial console connection.</p> <p>XMODEM SEND will send the current program held in the program memory to the remote device. XMODEM RECEIVE will accept a program sent by the remote device and save it into the S32K1xx program memory overwriting the program currently held there. Note that the data is buffered in RAM which limits the maximum program size.</p> <p>The CRUNCH option works like RECEIVE but it instructs MMBasic to remove all comments, blank lines and unnecessary spaces from the program before saving. This can be used on large programs to allow them to fit into limited memory.</p> <p>SEND, RECEIVE and CRUNCH can be abbreviated to S, R and C.</p> <p>The XModem protocol requires a cooperating software program running on the remote computer and connected to its serial port. It has been tested on minicom running on Linux.</p> <p>After running the XMODEM command in MMBasic press: CTRL-A and then Z to enter the menu. Continue with S to send a file from computer to S32K1xx or R to receive a file from S32K1xx.</p> <p>The transfer can take up to 15 seconds to start and if the XMODEM command fails to establish communications it will return to the MMBasic prompt after 60 seconds and leave the program memory untouched.</p>
--	---

# Functions

Note that the functions related to communications functions (I<sup>2</sup>C and Output configuration) are not listed here but are described in the appendices at the end of this document.

Square brackets indicate that the parameter or characters are optional.

ft-RPI-sa specific functions are colored in orange.

ABS( number )	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and a positive number is returned).
ACOS( number )	Returns the inverse cosine of the argument 'number' in radians.
ASC( string\$ )	Returns the ASCII code (ie, byte value) for the first letter in 'string\$'.
ASIN( number )	Returns the inverse sine value of the argument 'number' in radians.
ATN( number )	Returns the arc-tangent of the argument 'number' in radians.
BIN\$( number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
CHR\$( number )	Returns a one-character string consisting of the character corresponding to the ASCII code (ie, byte value) indicated by argument 'number'.
CINT( number )	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COS( number )	Returns the cosine of the argument 'number' in radians.
COUNTER( number )	Return the number of pulses counted on Cx input. <number> reflects the number of channel, ranging from 1 to 4. Actually, the number can go up to 8 as the counter is implemented in two different ways. Counter 1 to 4 are sampled @1ms. As debouncing need to be performed, it requires a minimum of 4 consecutive identical samples until a valid value gets detected. As counting is done with a positive edge on the appropriate counter input, 2 valid values need to be detected – first low, second high. Therefore, the maximum frequency which can be measured on these inputs is 125Hz (two times 4 ms). Counter 5-8 reflect the same counter inputs 1 to 4 but these are implemented using the DMA feature of the MCU. This works w/o any interaction of any SW and is much faster. Frequencies up to 100kHz can easily be measured. There's one drawback of the DMA implementation: There is no debouncing performed. This means, if you have a simple mechanical switch attached to a counter input and read out the DMA counter, you'll see many more pulses counted with one single switch. All counters can be reset by applying COUNTER(x)=0. Only a „0“ can be written to a counter.

DATE\$	Returns the current date based on MMBasic's internal clock as a string in the form "DD-MM-YYYY". For example, "28-07-2022". The internal clock/calendar will keep track of the time and date including leap years. To set the date use the command DATE\$ =.
DEG( radians )	Converts 'radians' to degrees.
DIPSWITCH	Returns the value of the DIP-Switch implemented on the board. Converting the returned value to binary, each bit is reflecting the appropriate switch. Example: Binary 10011 (19 dec) => DIP-Switch 1,2 and 5 are set to „ON“
DISTANCE(<Trig>, <Echo>)	This allows to connect an ultrasonic distance sensor (e.g. HC-SR04) and returns the distance given in mm. <Trig> is the appropriate trigger channel (1..4) to be connected to one of the counter inputs C1 to C4. <Echo> Is the input (1..8) of the echo pulse provided on one of the input connectors I1..I8. Example: Print Distance(2,4) Triggers the ultrasonic sensor connected to C2 and receives the echo pulse on input I4. The result is the distance given in mm.
E_ANA( number )	Returns a 12bit digital value reflecting the voltage applied to the input <number>. The provided <number> range from 1 to 9 whereas 9 refer to the on-board potentiometer. The voltage reference of the used analog to digital converter (ADC) on the MCU is sourced by 3.3V. This means, a value of 0 represents 0V and 4095 represents 3.3V. To convert voltages above 3.3V, it's required to add a resistor in series and have the appropriate pull-down resistor enabled (see E_PULL command in appendix B) to have a voltage divider provided transposing the voltage to the range the ADC can handle.
E_DIG( number )	Returns a „1“ if the voltage level on input <number> is greater than 1.88V and a „0“ if it's below. To change the level of the switchover point, an additional resistor need to be connected into series and the appropriate pull-down resistor need to be enabled (see E_PULL command) to have a voltage divider provided, transposing the voltage down to 1.88V.
E_INP	The returned value is a combination of all 8 inputs whereas each bit is reflecting the appropriate input level. The same voltage level of 1.88V is used for this command as for E_DIG. Example: Binary 11010011 (211 dec) → Input 1,2,5,7 and 8 are logic high and input 3,4 and 6 are logic low
EXP( number )	Returns the exponential value of 'number', ie, ex where x is 'number'.

FIX( number )	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point. For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX() and INT() is that FIX() provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behavior is for Microsoft compatibility. See also CINT() .
HEX\$( number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY\$	Checks the console input buffer and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If the input buffer is empty this function will immediately return with an empty string (ie, "").
INSTR( [start-position,] string searched\$, string-pattern\$ )	Returns the position at which 'string-pattern\$' occurs in 'string-searched\$', beginning at 'start-position'. If 'start-position' is not provided it will default to 1. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern\$' is not found.
INT( number )	Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
LEFT\$( string\$, nbr )	Returns a substring of 'string\$' with 'nbr' of characters from the left (beginning) of the string.
LEN( string\$ )	Returns the number of characters in 'string\$'.
LCASE\$( string\$ )	Returns 'string\$' converted to lowercase characters.
LOG( number )	Returns the natural logarithm of the argument 'number'.
MAX( arg1 [, arg2 [, ...]] ) or MIN( arg1 [, arg2 [, ...]] )	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
MID\$( string\$, start ) or MID\$( string\$, start, nbr )	Returns a substring of 'string\$' beginning at 'start' and continuing for 'nbr' characters. The first character in the string is number 1. If 'nbr' is omitted the returned string will extend to the end of 'string\$'
OCT\$( number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).

PEEK(BYTE addr%) or PEEK(WORD addr%) or PEEK(INTEGER addr%) or PEEK(FLOAT addr%) or PEEK(VARADDR var) or PEEK(VAR var, ±offset) or PEEK( VARTBL, ±offset) or PEEK( PROGMEM, ±offset)	<p>Will return a byte or a word within the S32K1xx virtual memory space. BYTE will return the byte (8-bits) located at 'addr%' WORD will return the word (32-bits) located at 'addr%' INTEGER will return the integer (64-bits) located at 'addr%' FLOAT will return the floating point number (32-bits) located at 'addr%' VARADDR will return the address (32-bits) of the variable 'var' in memory.</p> <p>An array is specified as var().</p> <p>VAR, will return a byte in the memory allocated to 'var'. An array is specified as var().</p> <p>VARTBL, will return a byte in the memory allocated to the variable table maintained by MMBasic. Note that there is a comma after the keyword VARTBL.</p> <p>PROGMEM, will return a byte in the memory allocated to the program. Note that there is a comma after the keyword PROGMEM.</p> <p>Note that 'addr%' should be an integer.</p> <p>For backwards compatibility PEEK( hiword, loword ) is still accepted. In this case the address is specifies by 'hiword' which is the top 16 bits of the address while 'loword' is the bottom 16 bits.</p> <p>This command is for expert users only. The S32K1xx maps all control registers, flash (program) memory and volatile (RAM) memory into a single address space so there is no need for INP or OUT commands. The S32K1 Reference Manual lists the details of this address space.</p>
PI	Returns the value of pi.
RAD( degrees )	Converts 'degrees' to radians.
RIGHT\$( string\$, number-of chars )	Returns a substring of 'string\$' with 'number-of-chars' from the right (end) of the string.
RND( number ) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The RANDOMIZE command reseeds the random number generator.
<b>RTCALARM</b>	<b>Returns a 1 once an alarm was triggered by the RTC, otherwise a 0.</b>
SGN( number )	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN( number )	Returns the sine of the argument 'number' in radians.
SPACE\$( number )	Returns a string of blank spaces 'number' characters long.
SQR( number )	Returns the square root of the argument 'number'.

STR\$( number ) or STR\$( number, m ) or STR\$( number, m, n ) or STR\$( number, m, n, c\$ )	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c\$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR\$(123.456) will return "123.456" STR\$(-123.456) will return "-123.456" STR\$(123.456, 1) will return "123.456" STR\$(123.456, -1) will return "+123.456" STR\$(123.456, 6) will return " 123.456" STR\$(123.456, -6) will return " +123.456" STR\$(-123.456, 6) will return " -123.456" STR\$(-123.456, 6, 5) will return " -123.45600" STR\$(-123.456, 6, -5) will return " -1.23456e+02" STR\$(53, 6) will return " 53" STR\$(53, 6, 2) will return " 53.00" STR\$(53, 6, 2, "") will return "****53.00"
STRING\$( nbr, ascii ) or STRING\$( nbr, string\$ )	Returns a string 'nbr' bytes long consisting of either the first character of string\$ or the character representing the ASCII value 'ascii' which is an integer or float number in the range of 0 to 255.
TAB( number )	Outputs spaces until the column indicated by 'number' has been reached on the console output.
TAN( number )	Returns the tangent of the argument 'number' in radians.
TIME\$	Returns the current time based on MMBasic's internal clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". To set the current time use the command TIME\$ = "hh:mm:ss".
TIMER	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. The timer is reset to zero on power up or a CPU restart and you can also reset it by using TIMER as a command. If not specifically reset it will continue to count up forever (it is a 64 bit number and therefore will only roll over to zero after 200 million years).
UCASE\$( string\$ )	Returns 'string\$' converted to uppercase characters.
VAL( string\$ )	Returns the numerical value of the 'string\$'. If 'string\$' is an invalid number the function will return zero. This function will recognize the &H prefix for a hexadecimal number, &O for octal and &B for binary.



# Anhang A

## I<sup>2</sup>C Kommunikation

Der Inter Integrated Circuit (I<sup>2</sup>C oder auch IIC) Bus wurde von Philips (jetzt NXP), für den Datenaustausch zwischen Halbleiterbauteilen, entwickelt. Der Standard dafür kann in folgendem Dokument nachgelesen werden: [http://www.nxp.com/documents/user\\_manual/UM10204.pdf](http://www.nxp.com/documents/user_manual/UM10204.pdf)  
Es gibt 4 Befehle, die für den I<sup>2</sup>C Bus verwendet werden können:

I2C OPEN speed, timeout	Dies schaltet den I <sup>2</sup> C Bus im Master Mode ein. 'speed' ist ein Wert zwischen 10 und 400 (für Busgeschwindigkeit zwischen 10 kHz bis 400 kHz). Für einige Geschwindigkeiten ist es nicht möglich den exakten Wert einzustellen (z.B. 390 wird auf 400 und auch andere Werte werden auf den nächsthöheren Wert aufgerundet). 'timeout' ist ein Wert in Millisekunden nach dem ein Transfer abgebrochen wird, sobald kein erfolgreicher Empfang bzw. Senden stattgefunden hat. Der kleinste Wert beträgt dafür 100. Ein Wert von 0 schaltet den Timeout komplett ab, obwohl dies hier nicht empfohlen wird.
I2C WRITE addr, option, sendlen, senddata [,senddata ....]	Sendet Daten an einen I <sup>2</sup> C Slave. 'addr' ist die Adresse des I <sup>2</sup> C Slaves. 'option' ist eine Zahl zwischen 0 und 3 (normalerweise ist diese 0) 1: Behält die Kontrolle über den Bus nach diesem Befehl (es wird kein STOP Bit nach einem Transfer gesendet) 2: Die Adresse wird als 10-bit Adresse behandelt 3: Kombiniert 1 und 2 (behält Buskontrolle und verwendet 10-bit Adresse). 'sendlen' ist die Anzahl der zu übertragenden Bytes. 'senddata' ist das zu übertragende Datum - dieses kann auf mehreren Arten spezifiziert werden, wobei alle Werte zwischen 0 und 255 liegen: <ul style="list-style-type: none"><li>• Die Daten können als individuelle Bytes auf der Kommandozeile angegeben werden: Beispiel: I2C WRITE &amp;H6F, 0, 3, &amp;H23, &amp;H43, &amp;H25</li><li>• Die Daten können in einem eindimensionalen Array vorliegen, ohne Angabe der Tiefe in Klammern (Start immer mit dem ersten Element): Beispiel: I2C WRITE &amp;H6F, 0, 3, ARRAY()</li><li>• Die Daten können ein String sein - aber kein const Typ: Example: I2C WRITE &amp;H6F, 0, 3, STRING\$</li></ul>
I2C READ addr, option, rcvlen, rcvbuf	Empfängt Daten vom I <sup>2</sup> C Slave. 'addr' ist die Slave's I <sup>2</sup> C Adresse. 'option' ist eine Zahl zwischen 0 und 3 (normalerweise ist diese 0) 1: Behält die Kontrolle über den Bus nach diesem Befehl (es wird kein STOP Bit nach einem Transfer gesendet) 2: Die Adresse wird als 10-bit Adresse behandelt 3: Kombiniert 1 und 2 (behält Buskontrolle und verwendet 10-bit Adresse). 'rcvlen' ist die Anzahl der zu übertragenden Bytes. 'rcvbuf' ist die Variable in der die empfangenen Daten gespeichert werden - diese kann folgendermassen definiert sein: <ul style="list-style-type: none"><li>• Eine String Variable. Bytes werden sequentiell in dem String gespeichert.</li></ul>

	<ul style="list-style-type: none"> <li>• Ein eindimensionales Array von Zahlen ohne Angabe der Tiefe. Empfangene Bytes werden sequentiell im Array abgelegt, beginnend mit dem ersten Element. Beispiel: I2C READ &amp;H6F, 0, 3, ARRAY()</li> <li>• Eine einfache numerische Variable (in diesem Fall muss rcvlen=1 sein!).</li> </ul>
I2C CLOSE	Schaltet das I2C Interface ab.

Nach einem I2C Transfer wird der Status in der Systemvariable MM.I2C gespeichert:

0 = Kein Fehler aufgetreten

1 = Ein NACK (**No ACK**nowledge) wurde empfangen

2 = Timeoutfehler

### 7 und 8 Bit Adressierung

Die Standardadressen für I2C verwendet 7-bit Adressierung (ohne R/W-bit).

MMBasic passt automatisch, je nach Transferrichtung, das R/W Bit an.

Einige Halbleiterhersteller geben eine 8-bit Adresse an, die das R/W-Bit beinhaltet. Dies kann festgestellt werden, wenn unterschiedliche Bereiche für Schreib- bzw. Lesebereiche angegeben werden. In diesem Fall dürfen nur die oberen 7 Bits der Adresse verwendet werden.

Beispiel: Wenn als zu lesende Adresse 0x9b und die zu schreibende Adresse als 0x9a angegeben ist, nimmt man die oberen 7 Bits der Adresse, welche somit zu 0x4d wird. Einfacher ist es, die angegebenen Adressen einfach durch 2 zu teilen oder den Wert einmal um eine Bit Position nach rechts zu schieben.

Eine andere Methode der Identifikation besteht darin, den Wert der Adresse zu überprüfen. Dieser sollte innerhalb des Bereiches von 0x08 bis 0x77 liegen, andernfalls hat der Hersteller eine 8-bit Adresse angegeben.

(Anmerkung: Adressen im Bereich 0000xxx (0x0-0x7), sowie 1111xxx (0x78-0x7f) sind reserviert)

### 10 Bit Adressierung

Die 10-bit Adressierung ist kompatibel zur 7-bit Adressierung, um es zu ermöglichen beiderlei Varianten innerhalb eines Busses zu verwenden.

Bei der 10-bit Adressierung werden immer 2 Adressbytes verwendet, wobei das erste ein spezielles Muster verwendet, um anzuzeigen, dass 10-bit Adressierung verwendet wird. Dies wird von MMBasic automatisch gemacht, sobald die entsprechende Option verwendet wird. 10-bit Adressen können in dem Bereich von 0x0 bis 0x3ff liegen.

# Appendix B

## Steuerung der Ausgänge

Der ft-RPI-sa Controller bietet 8 Ausgänge die von 4 vollständigen H-Brücken angesteuert werden. Diese H-Brücken können einfache Bürstenmotoren, aber auch Schrittmotoren ansteuern. Weiterhin ist es möglich, die H-Brücken zu entkoppeln, damit jeder der 8 Ausgänge individuell angesteuert werden kann. Damit ist es möglich, über jeden Ausgang Leuchten, Elektromagnete, Ventile oder Motoren in eine Drehrichtung anzusteuern.

Output Conf <device>, <mode>	<p>Konfiguriert die H-Brücke/ &lt;device&gt; ist entweder eine 1 oder 2, wobei 1 die Ausgänge 1-4 und 2 die Ausgänge 5-8 kontrolliert. &lt;mode&gt; ist entweder eine 1 oder 3, wobei eine 1 für die volle Kontrolle eines Bürsten- oder Schrittmotors verwendet wird. Dieser Modus erlaubt es einen Bürstenmotor im Rechts- bzw. Linkslauf zu betreiben, sowie ausgleiten oder abrupt anhalten zu lassen. Ein Wert von 3 entkoppelt die H-Brücken und jeder Ausgang kann unabhängig, für Leuchten, Elektromagnete oder Bürstenmotor in eine Drehrichtung, verwendet werden.</p>
Output Write <Value>	<p>Schreibt den Wert &lt;value&gt; in die H-Brücken. Jedes Bit entspricht dabei einem Ausgang (d.h. Bit 0 kontrolliert Ausgang 1, Bit 1 den Ausgang 2 etc.) Dies ist nur gültig und anwendbar bei H-Brücken, die mit dem Mode 3 (unabhängig) konfiguriert sind und wird von einer H-Brücke, die im Mode 1 konfiguriert ist, ignoriert. Falls keine H-Brücke im Mode 3 konfiguriert ist, wird eine Fehlermeldung ausgegeben. Ein setzen eines Bits auf "1", bedeutet, dass am entsprechenden Ausgang 9V getrieben werden. Dementsprechend bedeutet eine "0" am Ausgang 0V. Beispiel: Output Write 0x23 0x23 = 0b00100011 =&gt; Ausgänge 0,1 und 5 werden auf 9V gesetzt, alle anderen auf 0V, wenn beide H-Brücken im Modus 3 konfiguriert sind.</p>

<p>Output Set &lt;Output&gt;, [ON OFF percentage]</p>	<p>Setzt einen oder mehrere Ausgänge, angegeben in &lt;Output&gt;, entweder AN (=ON), AUS (=OFF) oder auf einen bestimmten relativen Wert zur Versorgungsspannung (9V). Dieses Kommando setzt voraus, dass die entsprechende H-Brücke im unabhängigen Modus (d.h. Mode 3) konfiguriert ist. Jedes Bit im angegebenen &lt;Output&gt; Wert, bezieht sich auf den entsprechenden Ausgang (Bit 0=Ausgang 0 etc.) Beispiel:</p> <ul style="list-style-type: none"> <li>• Output Set 4+64,ON</li> </ul> <p>Dies schaltet Ausgang 3 und 7 auf 9V (=AN/ON)</p> <ul style="list-style-type: none"> <li>• Output Set 255,20</li> </ul> <p>Dies schaltet alle Ausgänge auf einen Wert 20% von 9V (=9V*0.2=1.8V)</p> <ul style="list-style-type: none"> <li>• Output Set 32,OFF</li> </ul> <p>Ausgang 6 wird abgeschaltet (=AUS/OFF)</p>
<p>Output Motor &lt;Output&gt;,&lt;Direction&gt;[,&lt;Level&gt;]</p>	<p>Dieses Kommando benötigt eine H-Brücke, die im Mode 1 konfiguriert ist. &lt;Output&gt; ist die Nummer der entsprechenden H-Brücke und liegt im Bereich 1 bis 4. H-Brücke 1 und 2 steuern die Ausgänge 1 &amp; 2, sowie 3 &amp; 4. H-Brücke 3 und 4 steuern die Ausgänge 5 &amp; 6, sowie 7 &amp; 8. &lt;Direction&gt; variiert zwischen 1 bis 4 und ist definiert als:</p> <ol style="list-style-type: none"> <li>1: Motor dreht sich links oder rechts (abhängig von der Polung)</li> <li>2: Motor dreht sich entgegengesetzt</li> <li>3: Abrupter Halt</li> <li>4: Motor gleitet aus</li> </ol> <p>&lt;Level&gt; ist der Ausgangspegel, angegeben in Prozent und ist nur erlaubt, wenn &lt;Direction&gt; entweder 1 oder 2 ist.</p> <p>Beispiele:</p> <ul style="list-style-type: none"> <li>• Output Motor 2,1,100</li> </ul> <p>Schaltet den Motor an den Ausgängen 3&amp;4, mit einem Pegel von 100%, an.</p> <ul style="list-style-type: none"> <li>• Output Motor 4,3</li> </ul> <p>Stopped den Motor an den Ausgängen 7 &amp; 8 schnell.</p> <ul style="list-style-type: none"> <li>• Output Motor 1,0,30</li> </ul> <p>Schaltet den Motor an den Ausgängen 1 &amp; 2, mit einem Pegel von 30%, an.</p>

<p>Output EncMotor &lt;Output&gt;,&lt;Counter Input&gt;,&lt;Steps&gt;,&lt;MinSpeed&gt;,&lt; MaxSpeed&gt;</p>	<p>Dieses Kommando benötigt eine H-Brücke, die im Mode 1 konfiguriert ist.          &lt;Output&gt; ist identisch mit dem gleichen Parameter, der für den "Output Motor" Befehl verwendet wird.          &lt;Counter Input&gt; ist die Nummer des verwendeten Zähleringangs C[1..4] an dem der Encodermotor angeschlossen ist.          &lt;Steps&gt; ist die Anzahl der durchzuführenden Schritte, die im Bereich <math>\pm 10000</math> liegen muss.          Die Drehrichtung des Motors hängt von der Polung ab. Eine negative Anzahl &lt;Steps&gt; dreht den Motor in die entgegengesetzte Richtung als eine positive Anzahl.          &lt;MinSpeed&gt;,&lt;MaxSpeed&gt; ist eine Zahl im Bereich 1-100 und bestimmt den prozentualen Anteil der Maximalspannung:          Sobald der Motor startet, beginnt dieser sich mit dem "MinSpeed" Pegel zu drehen und dieser Pegel erhöht sich bis zu dem "MaxSpeed" Pegel.          Sobald der Motor sich dem Ende der durchzuführenden Schritte nähert, verringert sich die Geschwindigkeit wieder bis "MinSpeed" erreicht wird, um den Motor exakt zu positionieren.          Falls "MinSpeed" zu hoch gewählt wurde und der Motor nicht auf die exakte Position erreichen konnte, wird eine Warnmeldung ausgegeben.          Falls "MinSpeed" zu niedrig gewählt wurde, kann, je nach Last, die exakte Position nicht erreicht werden, da der Motor sich evtl. überhaupt nicht mehr dreht.</p> <p>Beispiel: Output EncMotor 2,2,1000,20,100          Der Motor an den Ausgängen 3 &amp; 4 sowie C2 wird mit 20% Pegel gestartet und mit einem maximalen Level von 100% für 1000 Schritte bewegt.</p>
<p>Output Stepper &lt;Motor&gt;,&lt;# Steps&gt;,&lt;Speed&gt;</p>	<p>Dieses Kommando benötigt eine H-Brücke, die im Mode 1 konfiguriert ist.          &lt;Motor&gt; ist die Nummer des anzusteuernenden Motors und liegt im Bereich 1 bis 4. Um einen Schrittmotor anzusteuern, sind 2 vollwertige H-Brücken notwendig. Somit kann der ft-RPI-sa maximal 2 Schrittmotoren ansteuern.          Die angegebene Motornummer ist kodiert als:</p> <ul style="list-style-type: none"> <li>1: Schrittmotor an Ausgang 1 bis 4 (Vollschritt)</li> <li>2: Schrittmotor an Ausgang 5 bis 8 (Vollschritt)</li> <li>3: Schrittmotor an Ausgang 1 bis 4 (Halbschritt)</li> <li>4: Schrittmotor an Ausgang 5 bis 8 (Halbschritt)</li> </ul> <p>&lt;# Steps&gt; ist die Anzahl der durchzuführenden Schritte und liegt im Bereich <math>\pm 10000</math>.          &lt;Speed&gt; ist die Frequenz, mit der die Schritte durchgeführt werden und ist limitiert durch den verwendeten Motor.          Üblicherweise sollten 200Hz kein Problem sein, der Bereich erlaubt jedoch 1-350Hz.</p>

Alle Ausgänge sind gegen Kurzschluss geschützt. Falls ein Kurzschluss erkannt wird, werden die entsprechenden Ausgänge abgeschaltet, die Status LED blinkt 3 mal mit ungefähr 3Hz und nach einer anschließenden Pause von 1.5s, wiederholt sich der Vorgang, bis der Kurzschluss für mehr als 100ms entfernt wurde.

Während der Kurzschluss signalisiert wird, läuft das MMBasic Program im ft-RPI-sa Controller weiter.



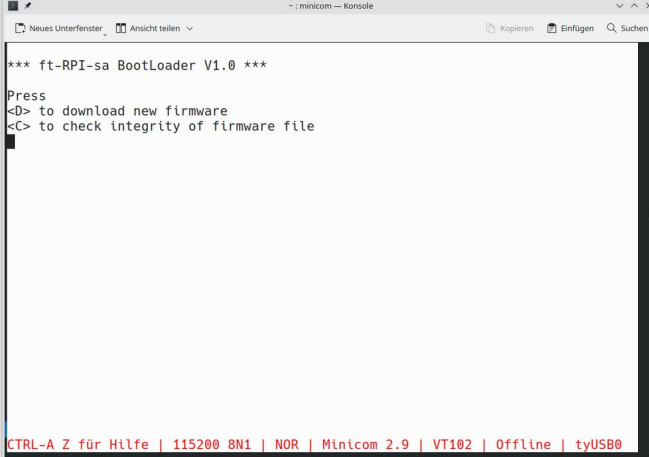
# Anhang C

## Firmware Update

Auf dem verbauten Mikrocontroller, einem S32K1xx Derivat, läuft das Programm (Firmware), dass den MMBasic Interpreter bereitstellt. Ab und zu werden Verbesserungen, Erweiterungen und Fehler beseitigt. Dazu muss das Programm auf dem Mikrocontroller ersetzt werden.

Prinzipiell ist dies möglich mit einem speziellen Interface, das auf den Anschluss J8 des ft-RPI-sa gesteckt wird. Dieses Interface ist relativ kostspielig und verlangt Know-How in der Handhabung sowie mit der Software Entwicklungsumgebung für die S32K1 Familie (Falls jemand Interesse haben sollte: [S32 Design Studio for S32 Platform](#)).

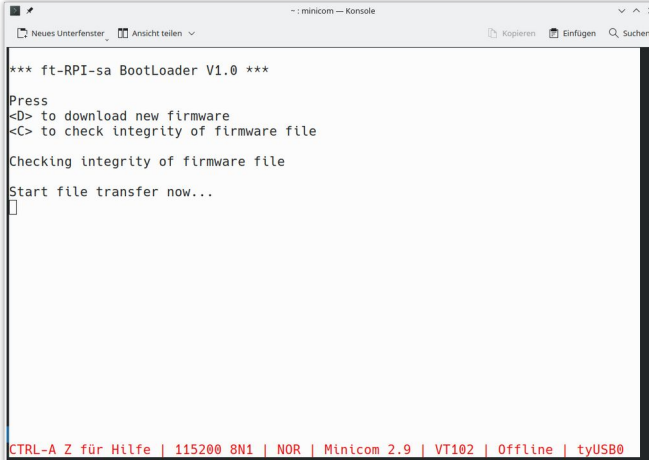
Eine weitere Möglichkeit besteht darin, dass der Mikrocontroller dieses Programm selbständig empfängt und seinen Speicher damit programmiert. Für diesen Fall ist ein spezieller Teil der Betriebssoftware bei einem RESET oder dem Einschalten zu starten. Damit der ft-RPI-sa auch weiß, wann dies zu geschehen hat, sind die beiden Anschlüsse JP5 und JP6 mit einer Kurzschlussbrücke zu schließen. Danach bekommt man bei drücken des RESET-Knopfes oder dem Einschalten folgendes zu sehen:



```
*** ft-RPI-sa BootLoader V1.0 ***
Press
<D> to download new firmware
<C> to check integrity of firmware file
```

CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0

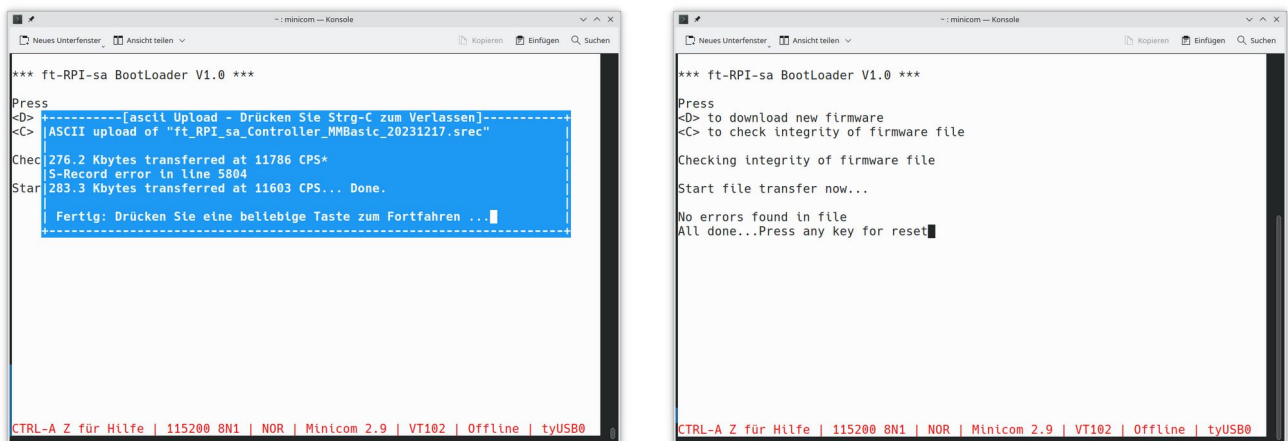
Bevor eine neue Firmware eingespielt wird, sollte diese unbedingt vorher überprüft werden, was mit drücken von "C" gestartet wird:



```
*** ft-RPI-sa BootLoader V1.0 ***
Press
<D> to download new firmware
<C> to check integrity of firmware file
Checking integrity of firmware file
Start file transfer now...
█
```

CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0

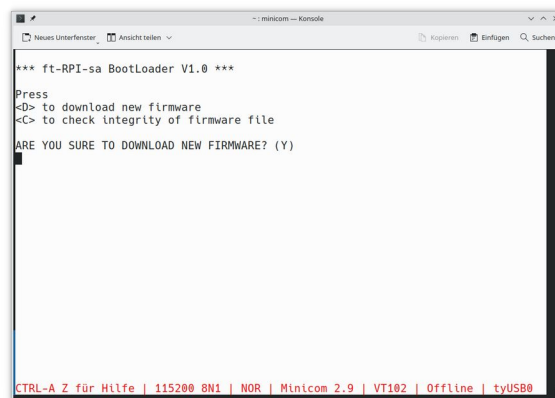
Nach der Überprüfung wird einem mitgeteilt, ob diese Firmware verwendet werden kann oder nicht:



```
*** ft-RPI-sa BootLoader V1.0 ***  
Press  
<D> -----[ascii Upload - Drücken Sie Strg-C zum Verlassen]-----  
<C> ASCII upload of "ft_RPI_sa_Controller_MMBasic_20231217.srec"  
Check 276.2 Kbytes transferred at 11786 CPS*  
S-Record error in line 5804  
Start 283.3 Kbytes transferred at 11603 CPS... Done.  
Fertig: Drücken Sie eine beliebige Taste zum Fortfahren ...  
  
CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0
```

```
*** ft-RPI-sa BootLoader V1.0 ***  
Press  
<D> to download new firmware  
<C> to check integrity of firmware file  
Checking integrity of firmware file  
Start file transfer now...  
No errors found in file  
All done...Press any key for reset
```

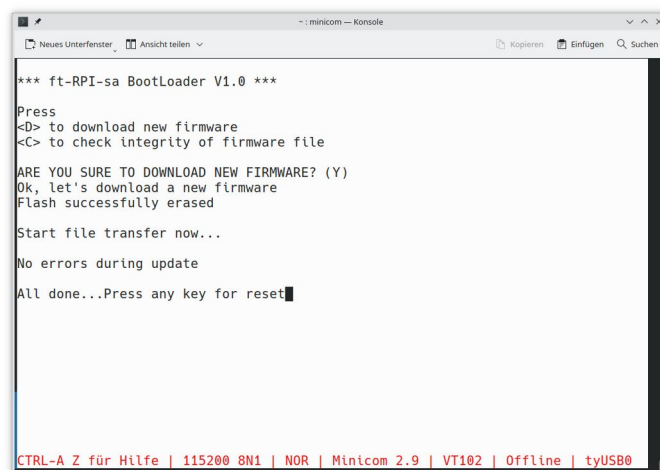
Wenn die Überprüfung erfolgreich war, kann die neue Firmware nun programmiert werden:



```
*** ft-RPI-sa BootLoader V1.0 ***  
Press  
<D> to download new firmware  
<C> to check integrity of firmware file  
ARE YOU SURE TO DOWNLOAD NEW FIRMWARE? (Y)  
|  
  
CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0
```

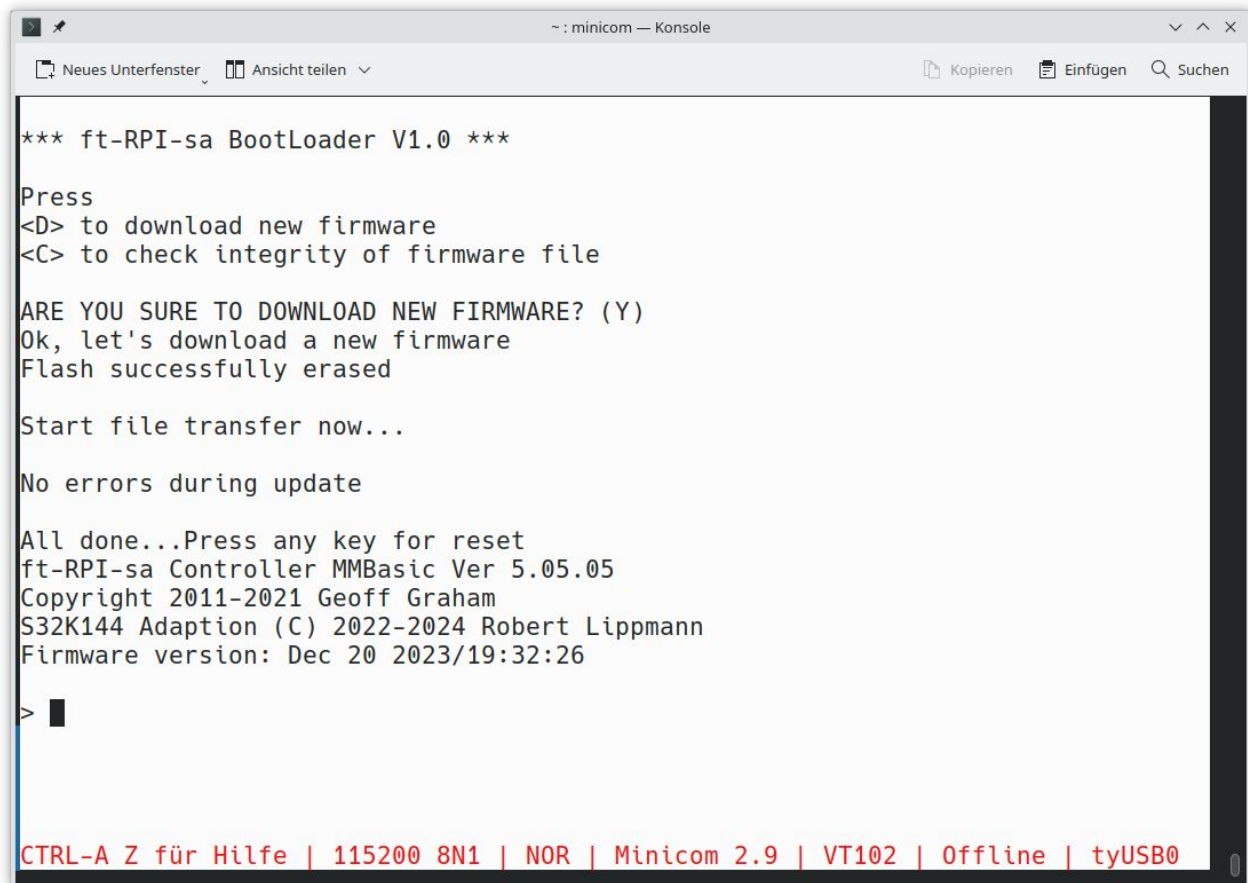
Jetzt ist der Moment gekommen, wo die aktuelle Firmware auf dem Mikrocontroller gelöscht wird. Falls nach dem drücken der "Y" Taste der Strom ausfällt oder die Verbindung zum PC/Raspberry PI unterbrochen wird, kann es passieren, dass der ft-RPI-sa danach nicht mehr gestartet werden kann. Dann kann nur noch jemand helfen, der das Eingangs erwähnte Spezialinterface besitzt!

Ansonsten läuft der Prozess eines Updates genauso ab, wie die Überprüfung:



```
*** ft-RPI-sa BootLoader V1.0 ***  
Press  
<D> to download new firmware  
<C> to check integrity of firmware file  
ARE YOU SURE TO DOWNLOAD NEW FIRMWARE? (Y)  
Ok, let's download a new firmware  
Flash successfully erased  
Start file transfer now...  
No errors during update  
All done...Press any key for reset
```

Nach dem erfolgreichen einspielen der neuen Firmware, kann nach dem Entfernen der beiden Kurzschlussbrücken JP5 & JP6, durch drücken einer beliebigen Taste die neue Version gestartet werden:



```
~ : minicom — Konsole
Neues Unterfenster Ansicht teilen
Kopieren Einfügen Suchen

*** ft-RPI-sa BootLoader V1.0 ***

Press
<D> to download new firmware
<C> to check integrity of firmware file

ARE YOU SURE TO DOWNLOAD NEW FIRMWARE? (Y)
Ok, let's download a new firmware
Flash successfully erased

Start file transfer now...

No errors during update

All done...Press any key for reset
ft-RPI-sa Controller MMBasic Ver 5.05.05
Copyright 2011-2021 Geoff Graham
S32K144 Adaption (C) 2022-2024 Robert Lippmann
Firmware version: Dec 20 2023/19:32:26
> █

CTRL-A Z für Hilfe | 115200 8N1 | NOR | Minicom 2.9 | VT102 | Offline | tyUSB0
```

# Anhang D

## Beispiel

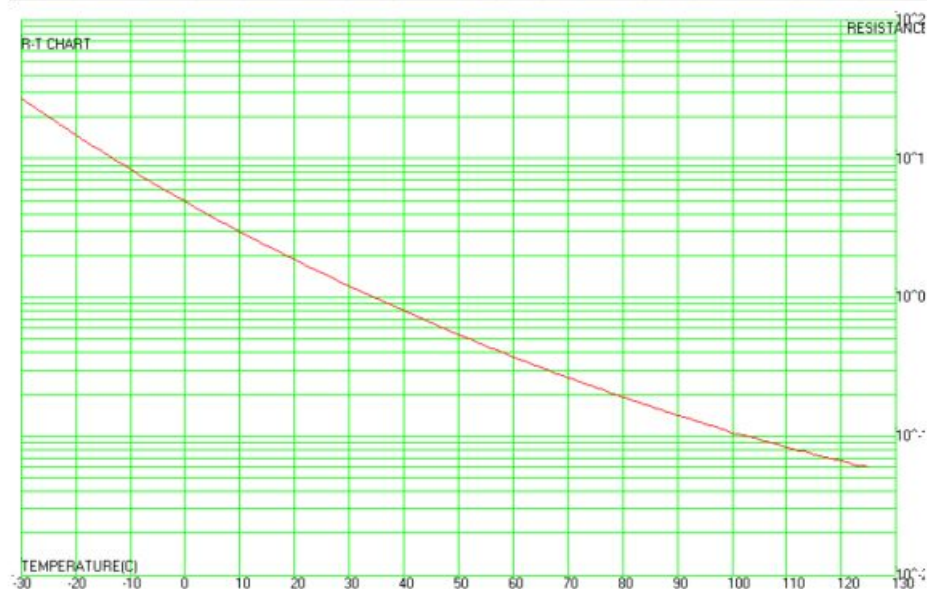
### Temperatur gesteuerter Lüfter

Zu verwendende Komponenten:



Datenblatt des NTC Widerstands um Widerstandswerte für bestimmte Temperaturen zu erhalten:

R25=1.5K $\Omega$						B25/50=3950K					
T	R	T	R	T	R	T	R	T	R	T	R
-30	27.36	-4	6.049	22	1.712	48	0.584	74	0.232	100	0.105
-29	25.7	-3	5.738	23	1.638	49	0.562	75	0.225	101	0.105
-28	24.149	-2	5.445	24	1.567	50	0.541	76	0.218	102	0.102
-27	22.697	-1	5.169	25	1.5	51	0.521	77	0.211	103	0.099
-26	21.339	0	4.935	26	1.435	52	0.502	78	0.204	104	0.097
-25	20.07	1	4.663	27	1.374	53	0.483	79	0.198	105	0.094
-24	18.882	2	4.431	28	1.316	54	0.466	80	0.192	106	0.092
-23	17.771	3	4.213	29	1.261	55	0.449	81	0.186	107	0.09
-22	16.731	4	4.006	30	1.208	56	0.433	82	0.18	108	0.088
-21	15.758	5	3.811	31	1.158	57	0.417	83	0.175	109	0.085
-20	14.847	6	3.627	32	1.11	58	0.402	84	0.17	110	0.083
-19	13.995	7	3.453	33	1.065	59	0.388	85	0.165	111	0.081
-18	13.197	8	3.288	34	1.021	60	0.375	86	0.16	112	0.079
-17	12.449	9	3.132	35	0.98	61	0.361	87	0.155	113	0.078
-16	11.748	10	2.985	36	0.94	62	0.349	88	0.15	114	0.076
-15	11.091	11	2.845	37	0.903	63	0.337	89	0.146	115	0.074
-14	10.475	12	2.712	38	0.866	64	0.325	90	0.142	116	0.072
-13	9.898	13	2.587	39	0.832	65	0.314	91	0.138	117	0.071
-12	9.356	14	2.468	40	0.799	66	0.304	92	0.134	118	0.069
-11	8.847	15	2.356	41	0.768	67	0.293	93	0.13	119	0.068
-10	8.369	16	2.249	42	0.738	68	0.283	94	0.127	120	0.066
-9	7.921	17	2.147	43	0.709	69	0.274	95	0.123	121	0.065
-8	7.499	18	2.051	44	0.682	70	0.265	96	0.12	122	0.063
-7	7.102	19	1.96	45	0.656	71	0.256	97	0.117	123	0.062
-6	6.73	20	1.873	46	0.631	72	0.248	98	0.113	124	0.061
-5	6.379	21	1.791	47	0.607	73	0.24	99	0.11	125	0.06



NTC Datenblattauszug

Programm:

Option default integer	‘ Standardtyp für Variablen ist Ganzzahl
EPull 2,2	‘ Pull am Eingang 2 ist ein Pull-Up
	‘ 'baut' einen Spannungsteiler mit dem NTC
Output conf 1,3	‘ Konfiguriert die H-Brücke für Ausgang
	‘ 1 bis 4 in den unabhängigen Modus
Const Udiv!=3.3/4096	‘ (!) für Gleitkommazahlen; berechnet
	‘ die ADC Auflösung pro Schritt bei 12bit
	‘ und 3.3V ADC Spannungsreferenz
Const Slope!=(18-40)/(2051-799)	‘ berechnet die Steigung der Kurve
	‘ zwischen 18°C und 40°C (Widerstand 2051Ω
	‘ und 799Ω aus dem Datenblatt)
Do	‘ Beginn einer Endlosschleife
MyInput=E_Ana(2)	‘ Spannung am NTC Widerstand holen
Intc!=(5-MyInput*Udiv!)/2700	‘ berechne den Strom durch den NTC: (5V
	‘ pull Spannung - Spannung am NTC)/Pull-up
	‘ Widerstandswert
MyTemp!=40+Slope!*(MyInput*Udiv!/Intc!-799)	‘ interpoliere zum
	‘ Temperaturwert

---

Formel für Interpolation erklärt:

$$f(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0} (x - x_0)$$

Quelle: Wikipedia

mit  $\frac{f_1 - f_0}{x_1 - x_0}$  als schon berechnete Steigung mit  $f_1=18^\circ\text{C}$ ,  $f_0=40^\circ\text{C}$ ,  $x_1=2051\Omega$  und  $x_0=799\Omega$

(Wert ist negativ, da die Kurve mit steigender Temperatur abfällt)

Es ergibt sich somit:

$$f(x) = f_0 + Slope! * (x - x_0)$$

mit x als der aktuelle NTC Widerstandswert, berechnet mit:

$$R_{NTC} = \frac{E_{ANA}(2) * U_{DIV!}}{I_{NTC!}}$$

---

If MyTemp!>25 Then	‘ Prüfe ob die Temperatur über Limit liegt
Output set 1,ON	‘ Schalte Lüftermotor ein
Else	‘ nein, sind unter dem Limit
Output set 1,OFF	‘ Schalte Lüftermotor aus
EndIf	
Loop	‘ wiederhole ab DO Befehl

---