

Aisle Reference Manual

Asher Hoskins

DRAFT

Aisle Reference Manual

Contents

1	Introduction	3
1.1	Grammar Syntax	3
1.2	Rigid and Flexible Aisle	3
1.3	Aisle for Dynamic Tesseract	4
2	Basic Definitions	4
2.1	Programs	4
2.2	Comments	4
2.3	Names	4
2.3.1	Defining Names	4
2.3.2	Namespaces	4
2.3.3	Canonical Names	5
2.3.4	Names in Inline Functions	5
2.3.5	Global Names	6
2.3.6	Differences Between Flexible Aisle and Rigid Aisle	6
2.4	Numbers	6
2.5	Variable Types	8
2.5.1	Void Pointers	8
2.5.2	Boolean Types	9
2.5.3	Type Definitions	9
2.5.4	Structures	9
2.5.5	Arrays	9
2.5.6	Unions	9
2.5.7	Enumerated Types	10
2.5.8	Layout of Variables in Memory	10
2.5.9	Information About Variables	10
3	Blocks	11
3.1	Statements	12
3.2	Block Offsets	12
3.3	Functions	12
3.3.1	Return Value	12
3.3.2	Inline Functions	13
3.4	Interrupts	13
3.4.1	Enabling and Disabling Interrupts	13
4	Memory Access	13
4.1	Program and Data Memory	13
4.2	Sources and Destinations	14
4.2.1	Structure and Union Access	15
4.2.2	Discarding Results	15

5	Variables, Aliases, and Constants	15
5.1	Variables	15
5.2	Aliases	15
5.3	Constants	16
6	Expressions and Assignments	16
6.1	Copy	16
6.2	Unary Expressions	16
6.3	Binary Expressions	17
7	Bulk Data Handling	17
7.1	Data	17
7.2	Table Lookup	18
7.3	Memory Allocation and Deallocation	18
7.4	Copying Multiple Bytes at a Time	19
8	Program Flow	19
8.1	Jumps	19
8.2	Function Calls	19
9	System Control	20
9.1	Resetting and Sleeping	20
10	Native Code	20
10.1	Using Variables and Constants in Native Code	21
10.2	Using Specific Registers	21
11	Conditional Compilation	22
12	Tracing Aisle Programs	22

1 Introduction

Aisle is a programming language, designed as an intermediate language¹ for the Tesseract system, although it may be used for other purposes. Any number of different source languages can be used to define component implementations in Tesseract as long as they can be compiled down to Aisle. The syntax of Aisle is designed to make optimisation and code generation rather than programming easy, since it is rare that people will have to write it directly.

1.1 Grammar Syntax

The Aisle language is described using a simple formal grammar. The grammar is made up of “symbols”, in *this font*, and “literals”, in **this font**. Symbols are defined as follows, with alternative definitions on different lines:

```
symbol:
  definition
  alternative-definition
```

Where a symbol is defined as one of a list of narrow alternatives these are shown on a single line preceded by the phrase “one of”:

```
symbol: one of
  one two three four
```

The top level symbol in a grammar is shown underlined when it is defined:

```
top-level:
  everything-else
```

Symbols that are optional have an “opt” subscript, like *this_{opt}*. A list of one or more symbols is shown like *this_{list}*. These two notations are combined to show a list of zero or more symbols, like *this_{list opt}*. Where a literal can be any one of a number of single characters in a range it is shown with a dash separating the first and last characters in the range like this: 0–9.

Some symbols, such as *ascii-character*, are left undefined. Their definition (in this case any valid ASCII character) should be obvious from their name.

1.2 Rigid and Flexible Aisle

There are two varieties of Aisle: “Flexible Aisle” and “Rigid Aisle”. Flexible Aisle is a superset of Rigid Aisle and is the variety of Aisle that may be output by code generators and the Tesseract compiler itself. It is converted into Rigid Aisle by the Aisle optimiser and pre-processor. Code generators only need to understand the simpler Rigid variety.

Grammar rules that only apply to Flexible Aisle are marked with an icon as shown below. Any rule that is not marked with this icon is part of both Aisle varieties.

```
example:
  this-rule-for-both-aisle-varieties
  F this-rule-for-flexible-aisle-only
```

¹“Intermediate Language” → “I.L.” → “Aisle”. Sorry.

1.3 Aisle for Dynamic Tesseractæ

Aisle statements and syntax items that are required only for the dynamic version of Tesseractæ are displayed *greyed out*. These do not have to be implemented for the initial static-only release of Tesseractæ.

2 Basic Definitions

2.1 Programs

Aisle programs have no fixed structure, statements can come in any order. A program will be scanned several times during code generation and so objects may be defined in the source file after the point where they are referenced.

program:
*statement*_{list opt}

2.2 Comments

Anything following the characters “//” on a line is ignored unless they appear inside a string or name. Comments can appear at any point in an Aisle program and so are not considered as part of the format grammar.

2.3 Names

So that programmers using languages that are compiled down to Aisle do not have to have any knowledge of Aisle keywords, and so know which names to avoid, all names are enclosed in round brackets. A name may consist of any printable ASCII characters except for the closing round bracket. This and other special characters may be entered into the name using backslash escape sequences as shown in table 1.

name:
(*ascii-character*_{list})

2.3.1 Defining Names

Statements that define names need not occur before those names are used in the source file. This means that, for example, “(x)=8; int (x);” is just as valid as “int (x); (x)=8;”. It also means that there is no need for function prototypes as in some other programming languages.

2.3.2 Namespaces

Names are organised hierarchically in “namespaces”. A new namespace is defined for each new block (see section 3). A name defined outside the current namespace may be referred to using a “hierarchical name”: a list of names separated by “:” giving all the parents of the required name in order from left to right. All names inside a namespace must be unique.

full-hierarchical-name:
name
full-hierarchical-name :: *name*

```

block (x) {
  block (y) {
    block (z) {}
  }
}

```

Listing 1: The scope of names (1).

```

block (a) {
  dynamic { int (x); }
  dynamic { int (y); }
  block (b) {
    dynamic { int (x); }
  }
}

```

Listing 2: The scope of names (2).

For example, consider the set of nested blocks shown in listing 1. To access the block (z) from another block the hierarchical name $(x)::(y)::(z)$ must be used.

Each namespace inherits all of the names from the namespace(s) above it so that, for example, in listing 2 the variable $(a)::(y)$ can be referred to as either $(a)::(y)$ or $(a)::(b)::(y)$.

If a name is the same as one defined in a parent block then its definition will hide that in the parent block. In listing 2 any references to a variable (x) will refer to $(a)::(b)::(x)$, not $(a)::(x)$.

2.3.3 Canonical Names

Since namespaces inherit all the names in their parents there are frequently multiple ways to refer to a given name. A “canonical name” is the hierarchical name that was first able to refer to the name. It is the name of the block in which it was defined followed by the name itself. In listing 2, the following canonical variable names are defined: $(a)::(x)$, $(a)::(y)$, and $(a)::(b)::(x)$.

canonical-name:

full-hierarchical-name (subset)

Canonical names are used in Rigid Aisle in order to simplify the name finding process.

2.3.4 Names in Inline Functions

When an inline function (available in Flexible Aisle only) is expanded, any names will be converted to their canonical forms *after* expansion. In order to refer to names that are inherited from higher namespaces *before* the expansion (such as instance variables when compiling from Tesseræ) the `this` keyword must be used.

this-name:

`[F] this :: name`

When the inline function in listing 3 is inserted into a program `this::(counter)` will be expanded to `(private)::(counter)`.

```

namespace (private) {

    int (counter);

    inline function int (nextval) { } {
        this::(counter) = this::(counter) + 1;
        result = this::(counter);
    }
}

```

Listing 3: Names in Inline Functions.

```

interrupt (x) {}

namespace (y) {
    interrupt (x) {}
}

```

Listing 4: Global Names.

2.3.5 Global Names

Two types of name (block offset and interrupt names) are called “global” names. They have no namespace prefix regardless of which namespace they were defined in. For example, in listing 4 both the interrupt blocks have the same name, (*x*), and would therefore clash.

2.3.6 Differences Between Flexible Aisle and Rigid Aisle

The only hierarchical names that Rigid Aisle can use are canonical names. Flexible Aisle may use full hierarchical names (which will be converted to their canonical forms by the pre-processor) and the “**this**” keyword.

hier-name:
canonical-name
 full-hierarchical-name
 this-name

2.4 Numbers

Integers may be expressed in binary (prefix “0b”), octal (prefix “0”), hexadecimal (prefix “0x”), or decimal (no prefix). Floating point numbers may use “e” or “E” to identify an exponential suffix. Integers may have the suffix “u” or “U” to specify that they are unsigned. Character constants consist of a printable ASCII character or character escape (as shown in table 1) enclosed in single quotes.

number:
*number-sign*_{opt} *unsigned-number*
number-sign: one of
 - +

Name	ASCII	Value	Escape Sequence
Audible alert	BEL	7	\a
Backspace	BS	8	\b
Horizontal tab	HT	9	\t
Newline	NL	10	\n
Vertical tab	VT	11	\v
Form feed	NP	12	\f
Carriage return	CR	13	\r
Double quote	"	34	\"
Single quote	'	39	\'
Closing bracket)	41	\)
Backslash	\	92	\\
Octal code		ooo	\o, \oo, \ooo
Hexadecimal code		hh	\xh, \xhh

Table 1: Character escapes.

unsigned-number:

binary-integer

octal-integer

decimal-integer

hexadecimal-integer

character-constant

simple-float

exponential-float

binary-integer:

0b *binary-digit*_{list} *unsigned*_{opt}

binary-digit: one of

0 1

octal-integer:

0 *octal-digit*_{list} *unsigned*_{opt}

octal-digit:

0-7

decimal-integer:

*decimal-digit*_{list} *unsigned*_{opt}

decimal-digit:

0-9

hexadecimal-integer:

0x *hexadecimal-digit*_{list} *unsigned*_{opt}

hexadecimal-digit: one of

0-9 a-f A-F

unsigned: one of

u U

Type	Bytes	Notes
<code>char</code>	1	Character/8-bit integer (signed by default)
<code>byte</code>	1	8-bit integer (unsigned by default)
<code>short</code>	2	Integer (signed by default)
<code>int</code>	4	Integer (signed by default)
<code>float</code>	4	Single-precision floating point
<code>double</code>	8	Double-precision floating point

Table 2: Standard variable sizes.

character-constant:

`' ascii-character '`

simple-float:

`decimal-integer . decimal-integer`

exponential-float:

`simple-float exponential number-signopt decimal-integer`

exponential: one of

`e E`

constant:

`number`

`hier-name`

`variable-info`

2.5 Variable Types

All the non-pointer types in Aisle have fixed sizes, as shown in table 2, independent of the target architecture. Note that the endianness of variables *can* vary depending on the architecture. The size of pointer variables depends on the address range of the target architecture. All integer types are signed by default except for `byte`.

variable-type:

`variable-signopt signed-type is-pointeropt`

`no-sign-type is-pointeropt`

`hier-name is-pointeropt`

variable-sign: one of

`signed unsigned`

signed-type: one of

`char byte short int`

no-sign-type: one of

`void bool float double`

is-pointer:

`*`

2.5.1 Void Pointers

The type “`void *`” is used to hold the addresses of blocks or labels. The non-pointer type “`void`” is used to mark a function as having no return value.

2.5.2 Boolean Types

The “bool” variable type can only hold the values zero or one. The word “TRUE” may be used instead of one, and the word “FALSE” may be used instead of zero.

2.5.3 Type Definitions

New types may be created in terms of existing types. Aliases from a hierarchically named type to a *local-name* may also be created. Type aliases are considered to be equivalent and may be used as though they were the same type.

```
type-defn:
    typedef variable-defn name ;
    typedef alias local-name is hier-name ;

variable-defn:
    variable-type
    structure-defn
    union-defn
    enum-defn

local-name:
    name
```

2.5.4 Structures

Structures may be used to create composite types. The “.” operator is used to access structure parts.

```
structure-defn:
    struct { create-variablelist opt }

create-variable:
    variable-type name arrayopt ;
```

2.5.5 Arrays

Attaching a constant in square brackets to the end of a variable name will create a fixed size array of that variable. Variable sized arrays and multidimensional arrays are not permitted. The value of an array variable is the address of the start of the array in memory. Array variables may not be directly assigned to, the [...] operators must be used to assign to elements of the array only.

```
array:
    [ constant ]
```

2.5.6 Unions

A union is a list of variables that all occupy the same location in memory. Unions are accessed like structures, with the “.” operator.

```
union-defn:
    union { create-variablelist }
```

2.5.7 Enumerated Types

Enumerated types allow names to be given to signed integers. They are dealt with rather more strictly in Aisle than in C. A name from one enumerated type may not be used to set a variable of a different enumerated type. Enumerated type values may however be used as though they were normal signed integers within expressions.

```
enum-defn:  
enum { enum-vallist }
```

Each name in the type may be given an individual value. If a value is not given then a name will take the value of the previous name defined in the type plus one. If the first name in a type is not given a value then it will take the value zero.

```
enum-val:  
name set-enum-valopt ;  
  
set-enum-val:  
= constant
```

2.5.8 Layout of Variables in Memory

The elements of a structure are laid out in memory in the order in which they are defined. There is no guarantee that a structure will be contiguous: some systems may pad structures so that each element starts on a word boundary.

Consider the following type definitions to define two types, (*type_x*) and (*type_y*):

```
typedef struct {  
    int (a);  
    byte (b);  
} (type_x);  
  
typedef struct {  
    (type_x) (c) [2];  
    short (d);  
} (type_y);
```

If we define an array (*type_y*) (*z*) [3] then it will be laid out in memory as in figure 1 (assuming an architecture where structures are contiguous).

2.5.9 Information About Variables

The number of bytes that a single instance of a variable type occupies is given by the `sizeof` operator. For compound types like structures this may be more than the total of all the sizes of its sub-types since some code generators may have to make sure each sub-type is on a word boundary in memory. The `isconst` operator will evaluate to one if a name is a constant and zero otherwise.

```
variable-info:  
sizeof { variable-type arrayopt }  
[F] isconst { hier-name }
```

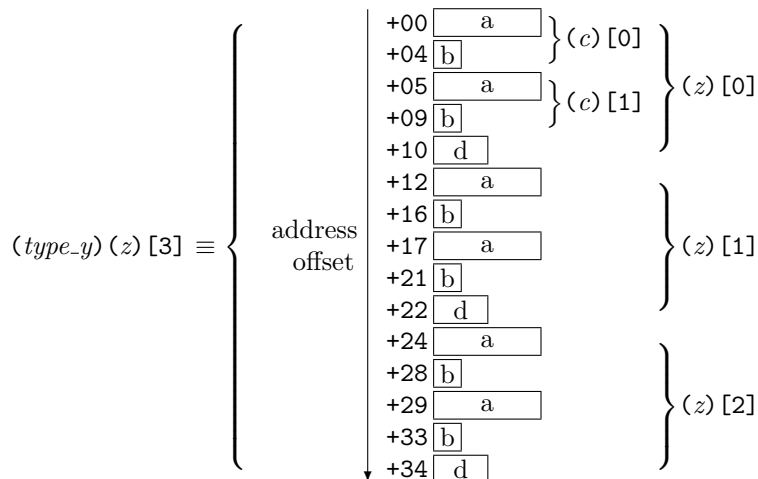


Figure 1: Layout of a Structure in Memory.

3 Blocks

A “block” of code is a series of statements surrounded by braces and marked with a “**block**”, “**namespace**”, “**function**”, or “**interrupt**” statement. **block** and **namespace** blocks may be nested. The simplest form of block is a **namespace**. This, as its name suggests, just starts a new namespace inside its parent block. A **block** is a little more complex. This marks a block of executable code and the optimiser may look for references to it to determine whether or not it can be safely removed in order to reduce the size of an executable program. The **function** block type is used to hold program functions and will return to its caller with an optional result value when its end is reached (execution in the other two blocks will run onto whatever follows when their end is reached). The **interrupt** block type is similar to **function** but is designed for holding interrupt service routines. No arguments are passed to an interrupt routine and a “return from interrupt” instruction will be executed at the end of the block.

A block name may be used in an expression as a pointer to program memory of type “**void ***”.

block:

block-type { *statement*_{list opt} }

block-type:

namespace *name*

block *name-or-offset*

*inline*_{opt} **function** *return-type* *name* { *create-variable*_{list opt} }

interrupt *exclusive*_{opt} *name*

name-or-offset:

name

offset *name* **from** *name*

inline:

inline

return-type:

variable-type

3.1 Statements

Each Aisle program and block is made up of a list of zero or more statements.

statement:
 type-defn
 create-var-const
 block
 assignment
 data-item
 free-storage
 copy-multiple-bytes
 jump
 function-call
 system-control
 native-block
 trace
 [F] *conditional-compilation*

3.2 Block Offsets

Adding “offset *x* from *y*” to a block definition will set the global name *x* to the byte offset of the block from the start of *y*, which must be a parent block.

Offset names may be reused and a reference to an offset will mark all the blocks that use it as “used” in the optimiser. If an offset is reused however it must be set to the same value as in its previous use. If its new value would be different then the Aisle code generator should exit with a fatal error.

3.3 Functions

A **function** must have a return type and can specify a list of arguments which will be accessible inside the function. These arguments will be passed by value to the function and, except in the case of inline functions, may be modified like normal variables.

Unlike **block** and **namespace** blocks, **function** blocks may not be nested inside other **function** blocks.

Use **void** as the *return-type* if the function does not return a value. The use of **result** inside such a void function will cause a compile-time error.

3.3.1 Return Value

Unlike C and other similar programming languages, functions in Aisle have only one exit point: at the end. The return value may be set using “**result**” as a *destination* in a statement. Multiple assignments to **result** may appear inside a function. Note that they will *not* cause the function to exit. At the end of a function block any **dynamic** variables created inside the function will be cleared up and the function will return to its caller.

result:
 result

3.3.2 Inline Functions

If a function is defined as “`inline`” then any calls to it will be replaced by the body of the function. The argument variables of an inline function may not be modified since they may get replaced with constants depending on how the function was called. Names will be converted to their canonical forms after an inline function call is expanded. The `this` keyword may be used to force canonisation before expansion (see also section 2.3.4).

There is no limit to the nesting of inline functions: inline functions may call other inline functions.

3.4 Interrupts

An `interrupt` block holds an interrupt service routine. The name is global and must match a system interrupt name as defined by the Aisle code generator that is being used. The processor state is saved automatically on entry to an interrupt service routine and restored on exit. Interrupt blocks may not be nested and there may be at most one interrupt block per system interrupt.

Interrupts may be interrupted by other interrupts unless the `exclusive` keyword is used when defining the interrupt block. Exclusive interrupts should be used with caution since they may interfere with vital system operations.

Since `interrupt` blocks end with a “return from interrupt” instruction they should not be called or jumped to like normal blocks. They should only be called by the processor’s interrupt hardware.

3.4.1 Enabling and Disabling Interrupts

The `interrupt enable` and `disable` statements are used to enable and disable interrupts. All interrupts are disabled by default.

Without a name these commands control the processor’s global interrupt flag which can enable or disable all previously enabled interrupts. When an interrupt is enabled by name the global interrupt flag will be automatically set to allow interrupts.

Note that some processors may have “non-maskable interrupts” which cannot be turned off. These will not be affected by any `interrupt` commands.

On processors with only a single interrupt the name is never needed.

```
interrupt-enable:
    interrupt enable nameopt ;

interrupt-disable:
    interrupt disable nameopt ;
```

4 Memory Access

4.1 Program and Data Memory

Aisle divides memory into two types: program memory, which is considered to be read-only; and data memory, which may be written to as well as read. All variables are created in and stay in data memory. To read from program memory the `table` and `progcpy` commands must be used (see sections 7.2 and 7.4).

4.2 Sources and Destinations

Aisle is strictly typed and so operations may only be performed on things that are the same type. There is no automatic type conversion, any type changes must be explicitly stated by prefixing with a type name inside braces. The only exception to this rule is when numeric constants are assigned to a variable. Here their type is converted automatically (making it valid, for example, to set both unsigned integer and floating point variables to the number “3” (if automatic type conversion did not occur in these cases then the programmer would have to use “3u” and “3.0” respectively)).

All type conversions are valid even when they could result in a loss of data (for example, converting from an `int` to a `byte`). Sign extension will occur only when converting to a larger signed type. This means that, for example, if (x) is a `signed byte` holding the value -1 (binary 11111111) then `int (x)` will be sign extended and so still hold the value -1 , but `unsigned int (x)` will not be sign extended and so will hold the value 255.

For the purposes of type conversions (they may actually be stored internally in a different format) all enumerated types are treated as `signed int`.

```
source:
    type-changeopt constant
    type-changeopt address-ofopt variable-access

address-of:
    &

destination:
    variable-access
    result
    discard

type-change:
    { variable-type }

variable-access:
    variable-element
    [--] variable-element
    variable-element [++]

variable-element:
    hier-name array-offsetopt var-element-partlist opt

var-element-part:
    . name array-offsetopt

array-offset:
    [ source ]
```

These variable accesses may be used to access data memory only. See section 7.2 and the `table` command for information on how access program memory.

When array offsets are used with pointer variables they are multiplied by `sizeof` the type that the pointer points to. The “&” operator may be used to get the address of a variable.

If a pointer variable is preceded by “[--]” (“pre-decrement”) then it will be decremented by `sizeof` the type it points to before the value it points to is fetched. If a pointer variable is followed by “[++]” (“post-increment”) then it will be incremented by `sizeof` the type it points to after the value it points to has been fetched.

```
typedef struct {
    int (a);
    int (b);
} (type x);

(type x) (x);
(type x) *(y);

(y) = &(x);
```

Listing 5: Structure Access via Pointer.

4.2.1 Structure and Union Access

The “.” operator is used to access parts of a structure or union. The same operator is used when accessing a structure or union via a pointer (unlike C, which uses the “->” operator in this case). Therefore, given the definitions in listing 5 the element (*a*) of (*x*) may be accessed as either (*x*).(*a*) or (*y*).(*a*).

4.2.2 Discarding Results

If `discard` is used as the destination of a statement then the statement will act as though executed (so pre/post decrementing/incrementing of variables will happen, function calls will occur, etc.) but the result will be thrown away.

```
discard:
    discard
```

5 Variables, Aliases, and Constants

```
create-var-const:
    create-storage
    create-alias
    create-constant
```

5.1 Variables

Dynamic (local) variables are created on the stack of a function and will be automatically deleted when the function returns (at the end of the function block). Variables created outside of a `dynamic` statement will be created statically in a fixed location in memory.

```
create-storage:
    dynamic { create-variablelist opt }
    create-variable
```

5.2 Aliases

An “alias” is a pseudo-variable. It allows a *local-name* and type to be used for an existing static variable or block address. It allows, for example, a `void *` pointer to point to a number of

structures of different types and still have structure element access work correctly by creating aliases for the pointer for each of the structure types.

The sizes of the original and aliased variable types must be the same. Aliases can point to other aliases as long as the chain of aliases is not circular.

create-alias:

```
alias variable-type local-name arrayopt is destination ;
```

Where the *destination* is a data memory access with automatic increment/decrement the increment/decrement will occur once only, when the alias is created, and not when the alias is accessed.

Note that the *destination* can be **result** or **discard** as well as a variable. In these cases any assignments to the alias variable are set as the function result or discarded respectively.

5.3 Constants

Scalar constants can be created using the “**const**” statement. To create constant arrays of data or constant structures use the “**data**” statement described in section 7.1. An attempt to create a constant from a value not known at compile time (such as the address of a **dynamic** variable, for example) will cause an error.

create-constant:

```
const variable-type name = source ;
```

6 Expressions and Assignments

Aisle is strictly typed and so all the variable types in an expression must be the same as the variable type of the object that is being assigned to. Type changes (for example, “**{int}**”) may be used to convert all variables to the same type.

assignment:

```
copy  
unary-expression  
binary-expression  
malloc-memory  
table-lookup
```

6.1 Copy

Copies the *source* to the *destination*. This statement can be used to copy whole structures too.

copy:

```
destination = source ;
```

6.2 Unary Expressions

The unary operators are described in table 3.

unary-expression:

```
destination = unary-op source ;
```

unary-op: one of

```
- ~ !
```

-	Negation.
~	Binary negation (ones-complement).
!	Logical NOT.

Table 3: Unary Operators.

*	Multiplication.
/	Division.
%	Modulus.
+	Addition.
-	Subtraction.
<<	Left shift.
>>	Right shift.
&	Binary AND.
	Binary OR.
^	Binary XOR.
&&	Logical AND.
	Logical OR.

Table 4: Binary Operators.

6.3 Binary Expressions

The unary operators are described in table 4.

binary-expressions:

destination = source binary-op source ;

binary-op: one of

** / % + - << >> & | ^ && ||*

7 Bulk Data Handling

7.1 Data

The `data` statement inserts arbitrary data into the compiled program code.

data-item:

`data variable-type constant-list ;`

`data variable-type string ;`

constant-list:

constant

constant-list , constant

string:

`" ascii-characterlist "`

Strings are enclosed in double quotes and may contain any printable ASCII character except the double quote. This and other special characters may be entered into the string using a

backslash escape sequence as shown in table 1. Strings are *not* automatically terminated by a null as in C, the programmer must explicitly supply the “\0” if it is required.

When a **data** statement is used with a structure type the order and types of the items in the *constant-list* must match the order and types of the elements of the structure, traversing it with a depth-first search. For example, this type:

```
typedef struct {
    char (c);
    int (i);
} (a.type);
```

could be used in a **data** statement as follows:

```
data (a.type) 'a', 1, 'b', 3, 'e', 89;
```

Note that when **data** is used with a structure the data stored in program memory will match how data would be stored in that structure in data memory. This means that on systems where structures are not necessarily contiguous there may be gaps. This means that **progcpy** can be used to initialise structures in data memory from fixed data in program memory.

7.2 Table Lookup

The **table** statement is used to access data in program memory. The first argument is the address of the start of the table (a `void *` block address) and the second is the byte offset from the start.

The amount of data that is read is set by the type of *destination* (i.e. `sizeof destination` bytes).

```
table-lookup:
    destination = table source , source ;
```

7.3 Memory Allocation and Deallocation

The **malloc** statement allows the creation of a block of memory of arbitrary length. Memory allocations do not always succeed and the action to take if the allocation fails must be specified on every **malloc**. An attempt to allocate a block of zero size will always fail.

Memory blocks may be deallocated using the **free** statement. If **free** is passed a value of zero or any invalid pointer value it will do nothing.

```
malloc-memory:
    destination = malloc source else malloc-fail ;

malloc-fail:
    reset
    goto hier-name

free-storage:
    free source ;
```

7.4 Copying Multiple Bytes at a Time

There are two Aisle statements that copy multiple bytes. `progcpy` copies from program memory to data memory and `datacpy` copies from data memory to data memory. The results of copying between regions of memory that overlap are undefined.

```
copy-multiple-bytes:
    copy-type to-addr , from-addr , num-to-copy ;

copy-type: one of
    progcpy datacpy

from-addr:
    source

to-addr:
    source

num-to-copy:
    source
```

8 Program Flow

8.1 Jumps

Note that for the relational operators, a value of zero is seen as FALSE and all other values as TRUE.

```
jump:
    jump-always
    jump-conditional

jump-always:
    goto hier-name ;

jump-conditional:
    if source relational-op source goto hier-name ;
    if !opt source goto hier-name ;

relational-op: one of
    < <= > >= == !=
```

8.2 Function Calls

The `call` instruction is used to call a function with an optional list of arguments. The `with` modifier allows one or more global variables to be set for the duration of the call only (i.e. they are set back to their original values when the function returns). Use `with` to set things like pointers to structures containing instance information when calling an object or component method.

Note that Aisle is strictly typed and so all function arguments must be of the correct type. Use type conversions in the arguments of the function call to change types that do not match.

```
function-call:
    return-valueopt call hier-name func-arg-listopt withopt ;

return-value:
    destination =
```

```

func-arg-list:
    func-arg
    func-arg-list , func-arg

func-arg:
    source

with:
    with { with-settinglist }

with-setting:
    name = source ;

```

9 System Control

```

system-control:
    reset
    sleep
    interrupt-enable
    interrupt-disable

```

9.1 Resetting and Sleeping

The `reset` statement causes a complete software reset of the system. Whether or not data is maintained over a reset depends on the system. The `sleep` statement causes the system to go to “sleep” until an interrupt is received and handled.

```

reset:
    reset ;

sleep:
    sleep ;

```

Note that after the system has been woken from `sleep` by an interrupt it will not automatically go back to sleep. Use the following code to sleep continually, waking only to handle interrupts.

```

block (doze) {
    sleep;
    goto (doze);
}

```

10 Native Code

Native blocks allow code native to the code generator to be inserted into the generated code. If the block is marked “`protect`” then the code generator’s optimiser will not change it. Blocks marked “`pre`” will be inserted at the start of the native code generated by the code generator, after any preceding `pre` blocks; blocks marked “`post`” will be inserted at the end of the generated native code.

```

native-block:
    protectopt native positionopt { native-optionlist opt }

```

```

position: one of
    pre  post

native-option:
    var-get
    var-put
    modify
    native-code

native-code:
    asm { native-statementlist opt }

```

Native code does not have to follow the same formatting conventions as Aisle but must have balanced braces, even if the braces are inside comments, since Aisle handling tools scan over native code by counting matching braces. Use the symbols “\{” and “\}” to insert unmatched braces. The backslash will be removed from the final code. Use two backslashes, “\\”, to insert a single backslash into the native code.

10.1 Using Variables and Constants in Native Code

Native blocks can access Aisle variables and constants using the **get** and **put** lists to specify which variables and constants the native code reads (**get**) and which variables it writes (**put**). The **const** operator allows Aisle constants to be referred to in the native code. The **reg** operator specifies which register a variable will be loaded to or from. When loading a register from a variable the number of registers to use must be specified.

```

var-get:
    get { native-var-get-constlist }

var-put:
    put { native-var-putlist }

native-var-get-const:
    const name = source ;
    reg name , constant = source ;

native-var-put:
    destination = reg name ;

```

When Aisle is being used on a processor with registers smaller than the variable size (for example, an 8-bit processor which would need four registers to hold an `int`) then variables will be spread over multiple registers. The order in which variables are split up (bigendian or littleendian) and the maximum number of variables that a native section can access depends on the code generator. For example, on an 8-bit AVR microcontroller if (`x`) is a 4-byte integer then the statement

```
get { reg (r16),4 = (x); }
```

will load registers R16–R19 with the value of (`x`).

10.2 Using Specific Registers

In some native code it may be necessary to modify certain registers (perhaps a required opcode will only work on a particular register). These registers must be listed in a **use** list.

```

modify:
    use { native-reglist }

native-reg:
    reg name ;

```

11 Conditional Compilation

The “**compif**” statement may be used to selectively compile portions of code. The code following the **compif** will only be compiled if the *const-expression* is TRUE, otherwise the code in the **else** section will be compiled.

```

conditional-compilation:
    [F] compif const-expression { statementlist opt } compif-elseopt

compif-else:
    [F] else { statementlist opt }

const-expression:
    constant relational-op constant
    !opt constant

```

12 Tracing Aisle Programs

Aisle is designed to be used as an intermediate language between a compiler and a code generator. If an Aisle program is compiled from a number of files in a higher level language it can be hard to work out which bit of Aisle code corresponds to which statements in the original program. The problem becomes much worse once the program has gone through the optimiser and sections have been inlined.

The “**trace**” statement is used to mark-up an Aisle program so that a source level debugger or file viewer can automatically determine the relationship between it and the original source file or files that it was generated from.

```

trace:
    trace block trace-type stringopt
    trace line trace-type stringopt

trace-type:
    string

```

A **trace block** sets the value of the key *trace-type* in the whole of the block that the **trace** statement appears as well as in any children of that block. A given key may be overwritten with a new value in a child block but must not be assigned multiple values within the same block. If *string* is missing then the key is created with a null value.

A **trace line** applies from the current line in the current block until the end of that block. It is not inherited by any children of the block.

Index

- blocks, 11
- bulk data, 17
 - copying, 19
 - defining, 17
 - in RAM, 18
 - reading, 18
- comments, 4
- conditional compilation, 22
- constants, 16
- expressions, 16
- flexible aisle only
 - `compif`, 22
 - conditional compilation, 22
 - `else`, 22
 - full hierarchical names, 6
 - `inline`, 11
 - inline functions, 13
 - `isconst`, 10
 - `this`, 5
- functions, 12
 - calls, 19
 - inline, 13
 - return value, 12
- grammar
 - syntax of, 3
- interrupts, 13
 - disabling, 13
 - enabling, 13
 - global interrupt flag, 13
- jumps, 19
- keywords
 - `!`, 16, 19
 - `!=`, 19
 - `(`, 4
 - `)`, 4
 - `*`, 8, 17
 - `+`, 17
 - `-`, 16, 17
 - `.`, 14
 - `/`, 17
 - `//`, 4
 - `::`, 4
 - `<`, 19
 - `<=`, 19
 - `==`, 19
 - `>`, 19
 - `>=`, 19
 - `[`, 9, 14
 - `[++]`, 14
 - `[--]`, 14
 - `%`, 17
 - `&`, 17
 - `&&`, 17
 - `^`, 17
 - `~`, 16
 - `]`, 9, 14
 - `<<`, 17
 - `>>`, 17
 - `|`, 17
 - `||`, 17
 - alias, 9, 16
 - asm, 21
 - block, 11, 22
 - bool, 8
 - byte, 8
 - call, 19
 - char, 8
 - `compif`, 22
 - `const`, 16, 21
 - data, 17
 - `datacpy`, 19
 - `disable`, 13
 - `discard`, 15
 - double, 8
 - dynamic, 15
 - `else`, 18, 22
 - `enable`, 13
 - enum, 10
 - exclusive, 11
 - `FALSE`, 9
 - float, 8
 - free, 18
 - from, 11
 - function, 11
 - `get`, 21
 - `goto`, 18, 19
 - `if`, 19

- inline, 11
- int, 8
- interrupt, 11, 13
- is, 9, 16
- isconst, 10
- line, 22
- malloc, 18
- namespace, 11
- native, 20
- offset, 11
- post, 21
- pre, 21
- progcpy, 19
- protect, 20
- put, 21
- reg, 21, 22
- reset, 18, 20
- result, 12
- short, 8
- signed, 8
- sizeof, 10
- sleep, 20
- struct, 9
- table, 18
- this, 5
- trace, 22
- TRUE, 9
- typedef, 9
- union, 9
- unsigned, 8
- use, 22
- void, 8
- with, 20

names, 4

- canonical, 5
- defining, 4
- global, 6
- hierarchical, 4
- in flexible aisle, 6
- in inline functions, 5
- in rigid aisle, 6
- namespaces, 4

native code, 20

numbers, 6

programs, 4

reset, 20

sleep, 20

statements, 12

strings, 17

tracing, 22

variables

- aliases, 15
- arrays, 9
- boolean, 9
- constants, 16
- creating, 15
- defining new types, 9
- enumerated types, 10
- layout in memory, 10
- structures, 9
- types, 8
- unions, 9
- void pointers, 8