# The Desi Intermediate Language, DIL

Brian Shearing

Peter Grogono

15 November 2015

This document specifies the Desi Intermediate Language, DIL—the textual interface between the Universal Desi compiler, UDC, and the Desi JIT compiler.

## Contents

## 1 Lexis

The lexical grammar of the DIL is as follows. It employs seven-bit ASCII characters only.

DIL = { *whitespace* | *command* | *operand* | EndOfLine } EndOfFile.  
*command* = (‘a’ .. ‘z’)$_{1..4}$ .  
*operand* = (‘G’ | ‘U’) *type* $\mathcal{W}$ | ‘L’ *type* ( $\mathcal{I}$ | $\mathcal{R}$ | $\mathcal{T}$ ) | $\mathcal{T}$.  
*type* = *domain* | ‘[’ *domain* *type* ‘]’.  
*domain* = ‘A’ | ‘B’ | ‘C’ | ‘E’ | ‘F’ | ‘O’ | ‘P’ | ‘R’ | ‘T’ | ‘W’ | ‘X’.  
$\mathcal{W}$ = *digit*$_+$ .  
$\mathcal{I}$ = [ ‘+’ | ‘-’ ] *digit*$_+$ | ‘0x’ *hex*$_+$ .  
$\mathcal{R}$ = [ ‘+’ | ‘-’ ] *digit*$_+$ ‘.’ { *digit* } [ (‘e’ | ‘E’) [ ‘+’ | ‘-’ ] *digit*$_+$ ].  
$\mathcal{T}$ = ‘"’ { *text-char* } ‘"’ | ‘’’ { *text-char* } ‘’’.  
*text-char* = *escape* | < character with ASCII code 32..126 >.  
*escape* = ‘\n’ | ‘\t’ | ‘\b’ | ‘\r’ | ‘\f’ | ‘\\’ | ‘\’’ | ‘\"’ | ‘\u’ *hex*$_4$ | ‘\U’ *hex*$_8$.  
*hex* = *digit* | ‘a’ .. ‘f’ | ‘A’ .. ‘F’.  
*digit* = ‘0’ .. ‘9’.  
*whitespace* = (Space | Tab)$_+$ .

The definitions of $\mathcal{I}$ and $\mathcal{R}$ are those of a Desi *Word* and *Real* respectively, except that whereas Desi allows underscores within numeric values the DIL does not. Boolean literals are represented by ‘0’ and ‘1’; that is, `LB0` for false, and `LB1` for true. Byte and Word literals are represented by integers, either decimal or hexadecimal. Texts may include UNICODE characters expressed as the ‘\uFFFF’ escape sequence, as in Java, or the ‘\UFFFFFFFF’ escape sequence, as in GO. Identifiers are expressed as texts, and hence may include UNICODE characters.

A DIL operand such as `UW1234` or `LT"xyz"` has a *kind* such as `U` (user-defined) or `L` (literal), and a *type* such as `W` (*Word*) or `T` (*Text*), as shown in the following table.

| | | A | B | O | C | E | P | X | R | F | T | W |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ADR | BOO | BYT | CEL | ENT | PRT | PRC | REA | RTN | TXT | WRD |
| G | GEN | - | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | - | $\mathcal{W}$ | - | $\mathcal{W}$ | - | $\mathcal{W}$ | $\mathcal{W}$ |
| L | LIT | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{R}$ | $\mathcal{W}$ | $\mathcal{T}$ | $\mathcal{I}$ |
| U | USR | - | $\mathcal{W}$ | $\mathcal{W}$ | $\mathcal{W}$ | - | $\mathcal{W}$ | - | $\mathcal{W}$ | - | $\mathcal{W}$ | $\mathcal{W}$ |

## 2 Syntax

The syntax of a DIL file is as follows:

| | | |
|---|---|---|
| DIL | = | { *line* } EndOfFile. |
| *line* | = | [ *statement* ] EndOfLine. |
| *statement* | = | *command* { *operand* }. |

Statements have an ordering determined by the following grammar, in which the name of a command stands for a complete statement. The phrase *other* represents any statement not mentioned explicitly elsewhere in the grammar.

| | | |
|---|---|---|
| DIL | = | { *component* }. |
| *component* | = | { *qualifier* } *heading* { *parameter* } *body* `end`. |
| *heading* | = | `cell` \| `prc` \| `rtn` \| `entr`. |
| *parameter* | = | `ipar` \| `chn` { `fld` } \| `opar`. |
| *body* | = | `ext` \| *block*. |
| *block* | = | `blk` { *declaration* } `edec` { *executable* } `eblk`. |
| *declaration* | = | `tmp` \| `var`. |
| *executable* | = | *qualifier* \| *invocation* \| *other* \| *block*. |
| *qualifier* | = | `file` \| `line` \| `lbl`. |
| *invocation* | = | `argb` { `put` } *call* { `get` } `arge`. |
| *call* | = | `call` \| `crtc` \| `crtx`. |

## 3 Statements

The following table shows each statement and the constraints on the types of its operands. For example, consider the statement:

| | |
|---|---|
| add $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ + $r$ |

The form of the statement is '`add` $p$ $q$ $r$'. The effect of the statement is that variable $p$ becomes set to the sum of $q$ and $r$. The result, $p$, may be a generated (`G`) or user-defined (`U`) variable of type `O`, `R`, or `W`; that is, *Byte*, *Real*, or *Word*. The values to be added, $q$ and $r$, may be generated, literal (`L`), or user-defined, and are of the same type as $p$. This is not the same as stating that $q$ and $r$ may be of type `O`, `R`, or `W`, which would permit $p$ to be *Real*, say, $q$ to be *Word* and $r$ to be *Byte*.

Code `D` denotes any valid scalar or map operand. It is employed in definitions of declarations such as `var` and `ipar` and also in generic statements such as `cpy`.

| | |
|---|---|
| add $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ + $r$ |
| and $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ and $r$ |
| argb $p$:L/W $q$:L/W | begin $p$ ins and $q$ outs |
| arge | end args of routine |
| ash $p$:GU/OW $q$:GLU/$p$ $r$:GLU/W | $p$ := $q$ >> $r$ |
| asst $p$:GLU/B $q$:GLU/T | unless $p$ then failure $q$ |
| blk | beginning of block |
| c $p$:GU/BORTW $q$:GLU/BOPRTW | $p$ := coerce $q$ |
| call $p$:L/A $q$:L/T | call $p$ |
| cat $p$:GU/T $q$:GLU/T $r$:GLU/T | $p$ := $q$ // $r$ |
| ceil $p$:GU/W $q$:GLU/R | $p$ = ceiling $q$ |
| cell $p$:L/T | begin cell; name $p$ |
| chn $p$:U/P $q$:L/W $r$:L/T | new chan $p$; fields $q$, name $r$ |
| cmp $p$:GLU/BOPRTW $q$:GLU/$p$ | $\mathcal{C}$ := $p$ c.f. $q$ |
| cpy $p$:GU/D $q$:GLU/$p$ | $p$ := $q$ |

| | |
|---|---|
| crtc $p$:L/A $q$:L/T $r$:L/W | create cell $p(..r)$; name $q$ |
| crtp $p$:GLU/P $q$:L/W $r$:L/T | new port $p$; fields $q$, name $r$ |
| crtx $p$:L/A $q$:L/T $r$:L/W | create prc $p(..r)$; name $q$ |
| div $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ / $r$ |
| eblk | end of block |
| edec | end of declarations |
| end | end of cell, prc or rtn |
| entr | begin main cell |
| eq $p$:GU/B $q$:GLU/BOPRTW $r$:GLU/$q$ | $p$ := $q$ = $r$ |
| ext $p$:L/T $q$:L/T | external: language $p$, name $q$ |
| fail $p$:GLU/T | failure $p$ |
| file $p$:L/T | File := $p$ |
| fld $p$:L/W $q$:L/T | field no. $p$; name $q$ |
| flor $p$:GU/W $q$:GLU/R | $p$ := floor $q$ |
| ge $p$:GU/B $q$:GLU/BORTW $r$:GLU/$q$ | $p$ := $q$ >= $r$ |
| get $p$:GLU/D $q$:L/W | $p$ := par $q$ |
| gt $p$:GU/B $q$:GLU/BORTW $r$:GLU/$q$ | $p$ := $q$ > $r$ |
| imp $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ implies $r$ |
| inv $p$:GU/OW $q$:GLU/$p$ | $p$ := invert bits of $q$ |
| ipar $p$:GU/D $q$:L/T | new input par $p$; name $q$ |
| ja $p$:L/A | goto $p$ |
| jeq $p$:L/A $q$:GLU/BOPRTW $r$:GLU/$q$ | goto $p$ if $q$ = $r$ |
| jf $p$:L/A $q$:GLU/B | goto $p$ if not $q$ |
| jge $p$:L/A $q$:GLU/BORTW $r$:GLU/$q$ | goto $p$ if $q$ >= $r$ |
| jgt $p$:L/A $q$:GLU/BORTW $r$:GLU/$q$ | goto $p$ if $q$ > $r$ |
| jle $p$:L/A $q$:GLU/BORTW $r$:GLU/$q$ | goto $p$ if $q$ <= $r$ |
| jlt $p$:L/A $q$:GLU/BORTW $r$:GLU/$q$ | goto $p$ if $q$ < $r$ |
| jne $p$:L/A $q$:GLU/BOPRTW $r$:GLU/$q$ | goto $p$ if $q$ <> $r$ |
| jt $p$:L/A $q$:GLU/B | goto $p$ if $q$ |
| jtb $p$:GLU/OW $q$:L/A | goto $q[p]$ |
| lbl $p$:L/A | label $p$ |
| le $p$:GU/B $q$:GLU/BORTW $r$:GLU/$q$ | $p$ := $q$ <= $r$ |
| len $p$:GU/W $q$:GLU/D | $p$ := #$q$ |
| line $p$:L/W | Line := $p$ |
| lsh $p$:GU/OW $q$:GLU/$p$ $r$:GLU/W | $p$ := $q$ << $r$ |
| lt $p$:GU/B $q$:GLU/BORTW $r$:GLU/$q$ | $p$ := $q$ < $r$ |
| mclr $p$:U/[BOTW,BOPRTW] | set $p$ to be empty |
| mcpy $p$:U/[$q$,$r$] $q$:GLU/BOTW $r$:GLU/BOPRTW | $p[q]$ := $r$ |
| mdel $p$:GU/BOTW $q$:U/[$p$,BOPRTW] | remove element $p$ from $q$ |
| mget $p$:GU/BOPRTW $q$:U/[$r$,$p$] $r$:GLU/BOTW | $p$ := $q[r]$ |
| mind $p$:GU/B $q$:U/[$r$,BOPRTW] $r$:GLU/BOTW | $p$ := $r$ in domain of $q$ |
| mpdi $p$:U/[$q$,BOPRTW] $q$:GU/BOTW | $q$ := iterator over $p$ |
| mpdt $p$:GU/B $q$:U/[$r$,BOPRTW] $r$:GLU/BOTW | $p$ := more of iter $q$ over $r$ |
| mpdu $p$:U/[$q$,BOPRTW] $q$:GU/BOTW | $q$ := next over $p$ |
| mpty $p$:GU/B $q$:U/[BOTW,BOPRTW] | $p$ := $q$ is empty |
| mul $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ * $r$ |
| nand $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ nand $r$ |
| ne $p$:GU/B $q$:GLU/BOPRTW $r$:GLU/$q$ | $p$ := $q$ <> $r$ |
| neg $p$:GU/ORW $q$:GLU/$p$ | $p$ := - $q$ |
| nop | skip |
| nor $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ nor $r$ |
| not $p$:GU/B $q$:GLU/B | $p$ := not $q$ |
| nrqy $p$:GU/B $q$:GLU/P $r$:L/W | $p$ := (non-blocking) rcv ready $q.r$ |
| nsqy $p$:GU/B $q$:GLU/P $r$:L/W | $p$ := (non-blocking) snd ready $q.r$ |
| opar $p$:GU/D $q$:L/T | new output par $p$; name $q$ |

| | |
|---|---|
| or $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ or $r$ |
| prc $p$:L/T | begin process; name $p$ |
| put $p$:GLU/D $q$:L/W | par $q$ := $p$ |
| rcv $p$:GU/D $q$:GLU/P $r$:L/W | $p$ := receive $q.r$ |
| rem $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ rem $r$ |
| rimp $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ reverse implies $r$ |
| rnd $p$:GU/W $q$:GLU/R | $p$ = round $q$ |
| rsh $p$:GU/OW $q$:GLU/$p$ $r$:GLU/W | $p$ := $q$ >>> $r$ |
| rsig $p$:GU/P $q$:L/W | receive signal $p.q$ |
| rtn $p$:L/T | begin rtn; name $p$ |
| snd $p$:GU/D $q$:GU/P $r$:L/W | send $p$ to $q.r$ |
| ssig $p$:GU/P $q$:L/W | send signal $p.q$ |
| sub $p$:GU/ORW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ - $r$ |
| tcpy $p$:GU/T $q$:GLU/W $r$:GLU/T | $p[q]$ := $r$ |
| tget $p$:GU/T $q$:GLU/T $r$:GLU/W | $p$ := $q[r]$ |
| tmp $p$:G/BORTW | new temp $p$ |
| tsec $p$:GU/T $q$:GLU/T $r$:GLU/W $s$:GLU/W | $p$ := $q[r..s]$ |
| tslc $p$:GU/T $q$:GLU/W $r$:GLU/W $s$:GLU/T | $p[q..r]$ := $s$ |
| var $p$:GU/D $q$:L/T | new var $p$; name $q$ |
| wait | yield |
| xor $p$:GU/OW $q$:GLU/$p$ $r$:GLU/$p$ | $p$ := $q$ xor $r$ |

## 4 Additional Consistency Rules

- There is precisely one entry.
- The operand that labels a cell, process, routine, or statement must be a literal address (kind L, type A).
- The labels of components are unique and labels within a component are unique.[1]
- Each call-statement refers to a defined routine, each crtc-statement to a defined cell, and each crtx-statement to a defined process.
- Each jump refers to a defined label within its component.

[1] The first realisation of the JIT has a stronger rule than this, namely that all labels across a compilation unit must be unique. Actually, the rule is even stronger: all integer qualifiers of operands across a compilation unit must be unique, whether qualifiers of variables or addresses. The JIT compiler exploits the fact that these numbers are derived from node numbers in the abstract syntax tree of the UDC, and are hence unique.