# Dado2 Llc User's Manual

Russell C. Mills

Columbia University
Computer Science Department

17 May 1990

CUCS-431-89

## Abstract

Llc is an extension of C for hierarchically parallel processing on distributed-memory parallel processors. The language has been implemented on Dado2, a massively parallel tree-structured MIMD multicomputer. This manual explains the features of the language as it is implemented on Dado2, its compilation, and execution. The manual complements the llc tutorial [2], the llc reference manual [3], and the llc report [4].



This research was conducted as part of the DADO project. It was supported in part by the New York State Science and Technology Foundation NYSSTF CAT(88)-5 and by a grant from Hewlett-Packard. The author is an AT&T Graduate Fellow.

# 1 Introduction

# 2 Dado Architecure

Dado is a massively parallel distributed-memory tree-structured multicomputer. In the Dado2 machine ( [5]), a complete binary tree of 8-bit processing elements functions as a coprocessor attached to a conventional host computer, the *principal processor.*

The current machine, Dado2, contains 1023 PEs connected in a complete binary tree. Each Dado2 PE consists of an 8-bit microprocessor (an Intel 8751), 64K bytes of RAM and 4K bytes of EPROM, and a semi-custom I/O chip designed to provide inter-PE communication. The Dado2 EPROM-resident kernel [1] manages inter-PE communication. The host controlling computer communicates with Dado2 through a standard interface such as an IEEE-488 (GPIB); from the point of view of the host operating system, Dado2 is a peripheral device. A separate communications interface connects the root PE to the host-Dado2 communication channel. This interface contains its own microprocessor and Dado2 I/O chip that communicate with the root PE, in addition to circuitry that communicates with the host computer. The interface is designed to make the Dado2 root PE the left child of the host computer.

Dado2 has two separate communication channels, each a byte wide: one through the processor ports and one through the I/O chip. The processor ports provide communication between a PE and its *tree neighbors* (parent, left and right children). The I/O chip provides global communication via a broadcast circuit and a resolve/report circuit (a multi-byte minimum-computing comparator). Dado2 communication cannot be interrupt-driven; PEs must anticipate and participate actively in all communication.

The Dado2 I/O chip handles communication through all or part of the Dado2 tree of PEs. A memory-mapped I/O chip register controls the connections of each I/O chip to its tree neighbors' I/O chips, so that any PE can disconnect itself from its parent or its children from itself. At any point in a program's execution, the I/O chip connections divide the Dado2 machine into a number of connected components. Within a connected component, all communication through the I/O chip is global and synchronous; all PEs in the connected component must participate in each communication.

Dado2 is a MIMD/SIMD multicomputer in the following sense. At any point in a program's execution, any PE (including the host computer) has a *retinue* of PEs that receive instructions from it. The principal processor's retinue is the entire set of Dado PEs. The instructions that a PE sends to its retinue are addresses in Dado code rather than machine instructions; each PE has its own stored program and executes in MIMD mode. But a PE cannot execute arbitrary code, since it must cooperate with other PEs in order to communicate. In the simplest mode of operation, the host computer repeatedly broadcasts Dado function pointers; all Dado PEs read the function pointers and execute the functions. A PE may also receive as a command the address of a function that causes it to send instructions for some time to its own retinue, a subset of its descendant PEs.

Dado's software architecture reflects its communication constraints. The Ilc language provides no explicit communications instructions: instead, the programmer specifies operations to be performed in parallel. The Ilc compiler translates these parallel operations into code for different PEs containing calls to kernel communication primitives. The language definition guarantees that for each communication, all PEs required to participate in that communication actually do so.

Since the host is a conventional computer, and Dado2 PEs are 8-bit microprocessors, data formats and sizes are different in the host and Dado2. The llc programmer need not be concerned about different formats, since the code generated by the llc compiler and the llc runtime library perform any necessary format translation, even for composite data items (**struct** and **union** types). Data sizes, however, are a matter for possible concern: **int** and **double** types typically have less precision in Dado PEs than in the host. The following table presents the sizes of atomic types in Dado2 llc:

- **char**
  1 byte, unsigned
- **short**
  2 bytes
- **int**
  synonymous with **short**
- **long**
  4 bytes
- **pointer**
  2 bytes (64K address space)
- **float**
  4 bytes (8-bit exponent, sign bit, 23-bit normalized mantissa)
- **double**
  synonymous with **float**

## 3 The llc Language

This section discusses the llc extensions to the C language. Each subsection describes a single construct or closely related set of constructs, and corresponds to a section in the llc reference manual [3]. The treatment here is informal and not free standing; it is meant to illuminate and complement the more formal discussion in the reference manual. For working llc programs incorporating these constructs, the user should turn to the llc tutorial [2].

### 3.1 Retinues and Evaluating Retinues

As mentioned above, associated with any Dado PE executing part of an llc program is a *retinue* of PEs that receive instructions from it, as well as an *evaluating retinue* of PEs actively executing those instructions. Each PE is the *director* of its retinue. The principal processor's retinue is the entire set of Dado PEs, the evaluating retinue is the entire retinue, and each Dado PE's retinue is empty. At any point in the execution of the program, an evaluating PE's retinue consists of descendant PEs that are not descendants of an evaluating descendant PE. In other words, when a PE P invokes parallel operations, all PEs higher in the partial ordering than P that are not invoking parallel operations themselves are available as retinue for those PEs that are invoking parallel operations, and the assignment of a given non-evaluating PE to an evaluating PE's retinue is based on the partial ordering in the obvious way. If an evaluating PE executes code that invokes parallel operations, its initial evaluating retinue for those operations is its entire retinue. Let's clarify this discussion with a picture. PP, the principal processor, has PEs N, P, Q, R, and S in its retinue, but its evaluating retinue consists of P and R. P and R execute a function f(), which invokes parallel operations. The P's retinue consists of the single PE Q, and R's retinue consists of S.

PP (principal processor)

N

P evaluates f()

Q in P's retinue while P evaluates f()

R evaluates f()

S in R's retinue while R evaluates f()

### 3.2 Declarations

Llc adds the ^ (*retinue*) unary operator (3.4) to the set of declarator-forming operators in C. A ^ operator used in a declarator declares its operand to be a *retinue-tuple*--an object of the specified type, but with one element in each retinue PE. For instance, the declaration

```
int ^i;
```

declares I to be a retinue-tuple of **Int**. If this declaration appears outside a function, **I** is visible for the rest of the file, and refers to an **Int** in each Dado PE. If the declaration appears in a block, **I** is visible only inside that block, and refers to an **Int** in each PE in the retinue of the PE executing the block. The ^ operator can be combined with other declarator-forming operators, but since the ^ operator is a unary operator, and binds more loosely than () and [], parentheses may be necessary to enforce the desired interpretation. Here are some examples of legal llc declarations and their interpretations:

```
int (^j)[5];      j is a
                      retinue-tuple of
                          array of 5
                              int

double ^f();      f is a
                      function returning
                          retinue-tuple of
                              double

void pmemcpy(char *^, char *, unsigned int);
                  pmemcpy is a
                      function of (
                          retinue-tuple of
                              pointer to
                                  char,
                          pointer to
                              char,
                          unsigned int
                      ) returning
                          void
```

Here are some illegal llc declarations:

```
struct s {
    char c;
    int ^i;              /* no retinue-tuple components */
};

char ^*s;                /* no pointers to retinue-tuples */

char **c;                /* no retinue-tuples of retinue-tuples */

int (^f)() {             /* no retinue-tuples of functions */
}
```

### 3.3 Par statement

The **par** statement, whose syntax is

**par** *retinue-statement*

invokes parallel execution of *retinue-statement* in the evaluating retinue of the retinue of the PE executing the code surrounding the **par** statement. *retinue-statement* can be any legal llc statement not containing **goto**. In the current llc implementation on Dado2, *retinue-statement* also cannot contain code for the retinue PE's retinues, such as **par** statements, although it can call functions that do. In other words, a function can have only one level of syntactic parallelism, though this is a restriction of the Dado2 implementation, not the language, and will be lifted soon.

A declaration of a retinue-tuple outside *retinue-statement* is equivalent (except for scope) to a declaration of a non-retinue-tuple inside *retinue-statement*.

### 3.4 ^ operator

The ^ (*retinue*) operator is the syntactic analog of the **par** statement, but since it is an operator,

^ *retinue-expression*

has a type and a value. On Dado2, the ^ operator can cause loss of precision when communicating **int** and **double** values from Dado PEs to the host.

### 3.5 Seq statement

A **seq** statement,

**seq** *director-statement*

embedded in a **par** statement causes the directing processor, rather than its retinue, to execute *director-statement* exactly once, provided that the evaluating retinue for the code containing the **seq** statement is nonempty.

### 3.6 !^ operator

The !^ (*sequential*) unary operator is the expression analog of the **seq** statement, but since it is an operator,

!^ *expression*

has a type and a value. The following example demonstrates how to use the values returned by a function that returns a retinue-tuple:

```
par {
    int ^f();           /* f returns a retinue-tuple of int */

    int i = !^f();      /* the directing PE calls f, which */
                        /* returns a value in each */
                        /* evaluating retinue PE */
}
```

A variation on the previous fragment brings up two interesting points about llc:

```
par {
    int ^f();           /* f returns a retinue-tuple of int */

    !^f();              /* the directing PE calls f */
                        /* return values are discarded */
    f();                /* each evaluating retinue PE calls f */
                        /* return values are discarded */
}
```

First, functions in llc, unlike storage, are accessible in any PE; the declaration of **f** makes **f** callable in the directing PE as well as its retinue. Second, this accessibility makes it impossible for the llc compiler to decide which PEs execute code without the help of the **!^** and **^** operators.

### 3.7 With statement

The **with** statement, with syntax

**with** (*retinue-expression*) *self-statement*

restricts the evaluating retinue for *self-statement* to the set of PEs where *retinue-expression* is true. Notice that *self-statement* is self code, not retinue code. This statement changes the director's evaluating retinue, but only the directing PE evaluates *self-statement*. Thus these statements set up the evaluating retinue. One can set up the evaluating retinue and activate it at the same time with the llc statement

**with** *retinue-expression* **par** *retinue-statement*

which is equivalent to

**par if** (*retinue-expression*) *retinue-statement*

### 3.8 :: operator

The **::** (*with*) operator is the expression analogue of the **with** statement:

*self-expression* **::** *retinue-expression*

The **::** operator modifies a processor's evaluating set during the evaluation of an expression. As with the **with** statement, the code so modified is self code, not retinue code, although typically it contains embedded retinue code. The **::** operator is useful in syntactic contexts requiring an expression instead of a statement. It is also useful in modifying the evaluating retinue a function inherits, since evaluating retinues are preserved across function calls.

### 3.9 ? operator

The llc **?** *select* unary operator singles out one processor from the evaluating retinue. The operand of the **?** operator is retinue code, and it returns a value in each evaluating PE in the retinue: 1 in one PE, and 0 in the others. The **?** operator is useful for arbitrarily selecting a single PE from a set satisfying some criterion. It also provides the only convenient means in llc for iterating over a set of data: see [2] for some programs that iterate over sets of PEs.

### 3.10 Reduction operators

Reduction operators in llc use a commutative, associative binary operator to combine a set of values in the evaluating retinue to produce a single value in the retinue's directing PE. See [3] for a list of llc's built-in reduction operators, and [2] for several programs that use them. User-defined reduction operators are not currently implemented, but will be soon.

### 3.11 Local and nonlocal functions and function calls

The llc compiler can often produce substantially better code if it knows that functions called from retinue code or from directing code do not invoke any parallel operations, or call functions that do.

For example, in the statement

```
par int i = f();
```

if **f** might invoke operations in its retinue, all PEs in the retinue must determine whether their children are in the current evaluating retinue, and detach any children that are evaluating **f**. That is, PEs not in the evaluating retinue join the retinue of the nearest ancestor that is. Since this is a relatively time-consuming operation, and since most programs invoke hierarchical parallelism infrequently, the llc compiler assumes unless it knows otherwise that functions called from retinue code are strictly local, that is, communicate with no other PE. One way the compiler can know that a function contains retinue code is for it to compile the function. But since separate compilation is the norm in llc, as in C, the language provides declaration syntax (the **nonlocal** keyword) to tell the compiler that a function is nonlocal, that is, may communicate with other PEs. see [3] for a discussion of the syntax. The llc compiler also has a switch that allows the programmer to force the compiler to assume that all functions called from retinue code are not strictly local.

Similarly, in the statement

```
par int i = !^f();
```

if **f** might invoke operations in its retinue, all retinue PEs must be prepared to accept instructions from the PE executing **f**. Again, this is a time-consuming operation. The llc compiler assumes by default that functions called in the directing PE may invoke parallel operations, since after all, the function containing the call to the function does. The language provides declaration syntax (the **local** keyword) to tell the compiler that a function is purely local.

## 4 Compiling and Linking

Once a program is written, the next step is to compile and link it. The *llcc* command is a generalization of the UNIX[1] *cc* command. *Llcc* preprocesses llc code with the C preprocessor, translates the result into C code for host and Dado PEs, and invokes host and Dado compilers, assemblers, and linkers to produce assembly, object, and executable files. *Llcc* accepts llc (*.llc*), C (*.c*), assembly (*.s*), object (*.o*), and archive (*.a*) files for host or Dado PEs and translates, compiles, assembles, and links them as directed.

When llcc processes an llc file, it creates a pair of C files distinguished by their suffix: *.c* for the host code and *.c51* for the Dado code, and at each subsequent processing stage, *llcc* produces another pair of files. *Llcc* treats each such pair of files as a unit. By default, *llcc* assumes that any file referred to on its

[1]UNIX is a trademark of AT&T Bell Laboratories

command line is an llc source file, or a (host, Dado) pair of files, described in the table below, and referred to by the host file name:

| llcc | host | Dado |
|---|---|---|
| C | *file*.c | *file*.c51 |
| assembly | *file*.s | *file*.s51 |
| object | *file*.o | *file*.o51 |
| archive | *file*.a | *file*.a51 |
| executable | *file* | *file*.e51 |

Since *llcc* treats each pair of files as a unit, the host file name can be used as a target in makefiles. It is also possible to inject host-only or Dado-only C, assembly, object, and archive files into the compilation process.

Recall that functions, once declared explicitly or implicitly, can be called in any PE, so conceptually at least, each function must be compiled for both host and Dado, and loaded into each Dado PE. But it would be wasteful to compile and load a function into Dado PEs that can be called only in the host. Since a typical llc program contains many such functions, *llcc* assumes by default that any function in an llc source file should be compiled only for the host, and provides two C preprocessor directives to override this assumption:

```
#pragma retinue
#pragma self
```

The **#pragma retinue** directive instructs *llcc* to compile the next function only for Dado PEs; the two directives together instruct *llcc* to compile both for host and for Dado PEs. It is up to the programmer to ensure that all functions called in host and Dado code actually get compiled and linked for host and Dado; undefined functions produce link-time error messages.

The llc translator converts an llc function into two C functions, one for the directing PE and one for the retinue PEs; the function in the directing PE controls the execution of the function in the retinue PEs by broadcasting code addresses that the retinue PEs read and jump to. The function in the directing PE retains its llc name, and can be called from llc, C, or assembly code, just as any C function can. The function in the retinue PEs is effectively anonymous, and can be called only by the corresponding function in the directing PE. Likewise, **extern** functions called from llc code can be defined in llc, C, or assembly code. Thus, a function compiled only for Dado PEs (#pragma retinue) and containing no retinue code is equivalent to the same function in a C file compiled only for Dado.

The translator converts a declaration of retinue-tuple storage of a type into a declaration in the retinue of storage of that type. The storage retains its llc name, and can be referred to in llc, C, or assembly code. Likewise, **extern** storage referred to in llc code can be defined in llc, C, or assembly code. Thus, a global retinue-tuple object declared in an llc file is equivalent to the same (non-retinue-tuple) object declared in a C file compiled only for Dado.

*Llcc* also has a compilation mode for llc files that converts llc code into host-only C code; except for differences stemming from word sizes and data formats, this C code executes in the host exactly as the original llc code executes in the host attached to a 1-PE Dado machine. This mode makes it possible to

debug most of an llc program using any debugger available on the host and without using a Dado machine. Also, compiling for the host alone is much faster than for host and Dado, since *llcc* does not need to invoke the Dado C compiler, assembler, or linker.

## 5 Library Functions

*Llcc* automatically links every program with a library (*libllc.a*) containing useful llc and C functions for host and Dado PEs. Below are ANSI C declarations for these functions and a brief description of each.

```
int gprintf(char *format, ...);
int eprintf(char *format, ...);
```

Dado PEs have a connection to **stdout** and **stderr** in the host PE through **gprintf()** and **eprintf()**. The set of conversion specifications supported is {*cdflsux*}.

```
int self();
int parent();
```

Each PE (including the host) has a unique ID given by a breadth-first numbering starting with 0 in the host. The **self()** function returns this number, while **parent()** returns the ID of a PEs parent (and returns 0 in the host).

```
void pmemcpy(char *^to, char *from, unsigned int length);
```

The **pmemcpy** function is a parallel **memcpy**. It copies **length** bytes from the directing PE to the evaluating retinue PEs. Because the host and Dado machines typically have different word sizes and data formats, calls to **pmemcpy** in the host should operate only on arrays of **char**.

```
char *^demand(char *name);
```

*Llcc* produces a single Dado executable file that gets loaded into all Dado PEs. In some programs it may be important to give different code to different PEs. The **demand** function provides run-time linking and loading of Dado code. When passed the name of a Dado symbol such as a function name, **demand()** searches a Dado object-code library built by the programmer and loads any new code required to resolve the symbol reference into the evaluating PEs at an address returned locally by **malloc**. **demand** then returns in each PE the symbol's address. In a typical application of **demand**, many functions are to be distributed one or more per PE, and are to be called locally through a function pointer initialized to the value returned by **demand**. It is up to the programmer to distribute the functions according to any appropriate criteria. For technical reasons, the functions in the Dado object library must be C functions, not llc functions; that is, they must not contain any parallel constructs.

```
void init_timer(void);
void zero_timer(void);
void start_timer(void);
unsigned long stop_timer(void);
unsigned long read_timer(void);
```

Monitoring parallel programs is even more important than monitoring sequential programs, since the major reason for writing them is execution speed, and since performance bottlenecks can be subtle and difficult to detect. This set of functions provides a simple interface to a stopwatch-style timer that measures time in microseconds. The resolution of the timer in Dado PEs is 1 microsecond, but on a typical UNIX host it is 10 milliseconds. The timer must be set up with a one-time call to **init_timer**; after that it can be started with **start_timer**, reset with **zero_timer**, read with **read_timer**, or stopped and read with **stop_timer**.

```
void init_clock(void);
void zero_clock(void);
void start_clock(void);
unsigned long stop_clock(void);
unsigned long read_clock(void);
```

This set of functions provides a second independent, functionally identical timer.

## 6 Running a Program

Finally, after a program compiles and links successfully, it's time to run it. The *llcrun* command runs an llc program. Command-line options specify which Dado machine to use, the number of Dado PEs to use (if not the entire machine), and whether to run the communications protocol that allows messages sent by Dado calls to **gprintf()** and **eprintf()** to reach the host.[2] *Llcrun* assumes that if *file* is the name of the host executable, *file.e51* is the name of the Dado executable, and *file.a51* is the name of the Dado object library searched in calls to **demand**. *Llcrun* loads the Dado executable into the specified number of PEs and **exec**s the host program. During program execution, the host communicates with the Dado PEs across a standard interface. Run-time routines on the host perform format conversion on data sent to and received from Dado.

[2]Not running this protocol causes each PE to discard any messages it generates.

# I. Efficiency Considerations

A number of factors affect how fast an Ilc program runs on Dado2, and how efficiently it uses the available processing power.

Although Ilc has synchronous semantics, the Dado2 implementation provides asynchronous computation and synchronous communication. Any synchronization reduces an Ilc program's efficiency, since the slowest PE holds back all the others, and because synchronization is effected through global communication, which itself takes time. Each Ilc operator that moves data between PEs enforced a synchronization, since the entire retinue, not just the evaluating retinue, must participate in the communication that implements the operator. Synchronization by Ilc operators is most pernicious in flow-of-control constructs (**if** and **switch**) in retinue code. If the various branches contain only local code, they can be executed by different PEs in parallel, but if they contain nonlocal function calls or Ilc operators, the branches must be executed sequentially, as described in [3]. Looping constructs (**while**, **for**, and **do**) that contain nonlocal code force a synchronization at each loop iteration.

Certain Ilc operations are well supported by the Dado2 hardware, while others are not. The compiler uses Dado2 I/O chip communication to implement the **!^** and **^** operators, so these are quite efficient: the I/O chip can move data upward or downward in Dado at about 50 kilobytes/second with no startup cost to slow down small transfers; the transfer rate is independent of the number of Dado2 PEs participating in the communication. The **?** operator and the reduction operators **min/**, **max/**, **||/**, and **&&/** are implemented with I/O chip communication, so these too are fast, but the other reduction operators, **+/**, ***/**, **|&**, **&/**, and **^/**, use processor-port communication, which takes time proportional to the number of levels in the director's retinue.

Moving a struct as a unit is faster than communicating each component separately. Packaging an array in a struct lets the compiler generate more efficient code to move the array.

The compiler and runtime library handle nonlocal function calls inside retinue code (hierarchical parallelism) as well as they can, given the language definition and the constraints of the Dado2 architecture, but in retinue code, a nonlocal function call is much more expensive than a local one. At the very least, a nonlocal function call is a synchronization point, while a local one is not. Careful use of the **local** and **nonlocal** keywords can minimize the synchronization and communication overhead.

Finally, the communications protocol that allows messages sent by Dado calls to **printf()** and **eprintf()** to reach the host adds time proportional to the number of levels in the Dado machine and to the total message length to each host-Dado communication.

## References

[1] Mills, R. C., Radouch, Z., and Maguire Jr., G. Q.
*A New Kernel for the DADO2 Parallel Computer.*
Technical Report, Department of Computer Science, Columbia University, June, 1987.

[2] Mills, R. C.
*A Tutorial Introduction to llc.*
Technical Report, Department of Computer Science, Columbia University, 1989.

[3] Mills, R. C.
*Llc Reference Manual.*
Technical Report, Department of Computer Science, Columbia University, 1989.

[4] Mills, R. C.
*The llc Parallel Language and its Implementation on DADO2.*
Technical Report, Department of Computer Science, Columbia University, 1989.

[5] Stolfo, S. J., and Miranker, D. P.
The DADO Production System Machine.
*Journal of Parallel and Distributed Computing* 3(2):269-296, 1986.