Contents

Fortran M Language Definition

Ian T. Foster K. Mani Chandy

Abstract

This document defines the Fortran M extensions to Fortran 77. It updates an earlier definition, dated June 1992, in several minor respects.

1 Introduction

The reader is referred to other reports for additional information on the Fortran M language [2], its theoretical foundations [1], and a Fortran M compiler developed at Argonne National Laboratory [3].

2 Syntax

Backus-Naur form (BNF) is used to present new syntax, with nonterminal symbols in *slanted* font, terminal symbols in `TYPEWRITER` font, and symbols defined in Appendix F of the Fortran 77 standard [4] <u>underlined</u>. The syntax `[`*symbol*`]` is used to represent zero or more comma-separated occurrences of *symbol*; `[`*symbol*`]`$^{(1)}$ represents one or more occurrences.

2.1 Process, Process Block, Process Do-loop

A *process* has the same syntax as a subroutine, except that the keyword `PROCESS` is substituted for `SUBROUTINE`, `INTENT` declarations can be provided for dummy arguments, and a process cannot take an assumed size array as a dummy argument.

A *process call* can occur anywhere that a subroutine call can occur. It has the same syntax as a subroutine call, except that the keyword `PROCESSCALL` is substituted for `CALL`. In addition, process calls can occur in process blocks and process do-loops, and recursive process calls are permitted. A *process block* is a set of statements preceded by a `PROCESSES` statement and followed by a `ENDPROCESSES` statement. A block includes zero or one subroutine calls, zero or more process calls, and zero or more process do-loops. A process do-loop has the same syntax as a do-loop, except that the `PROCESSDO` keyword is used in place of `DO`, and the body of the do-loop can contain only a process do-loop or a process call.

A port variable or port array element can be passed as an argument to only a single process in a process block or process do-loop, and then cannot be accessed in a subroutine called in that block.

2.2 New Declarations

Five new declaration statements are defined: `INPORT`, `OUTPORT`, `INTENT`, `PROCESSORS`, and `PROCESS COMMON`.

inport_declaration	::	`INPORT ( [`*data_type*`] ) [`*name*`]`$^{(1)}$
outport_declaration	::	`OUTPORT ( [`*data_type*`] ) [`*name*`]`$^{(1)}$
intent_declaration	::	`INTENT(IN) [`*name*`]`$^{(1)}$ |
		`INTENT(OUT) [`*name*`]`$^{(1)}$ |
		`INTENT(INOUT) [`*name*`]`$^{(1)}$
machine_declaration	::	`PROCESSORS(` *bounds* `)`
name	::	`variable_name` | `array_name` | `array_declarator`
data_type	::	*fortran_data_type* |
		fortran_data_type name |
		`INPORT ( [`*data_type*`] )` |
		`OUTPORT ( [`*data_type*`] )`

In the `PROCESSORS` statement, *bounds* has the same syntax as the arguments to an `array_declarator`. The product of the dimensions must be nonzero. Any program, process, subroutine, or function including a `LOCATION` or `SUBMACHINE` annotation must include a `PROCESSORS` declaration.

The symbol *fortran_data_type* denotes the six standard Fortran data types. The dimensions in an `array_declarator` in a port declaration can include variable declared in the port declaration, parameters, and arguments to the process or subroutine in which the declaration occurs. The symbol "`*`" cannot be used to specify an assumed size. Variables declared within a port declaration have scope local to that declaration.

A `PROCESS COMMON` statement has the same syntax as a `COMMON` statement.

2.3 New Executable Statements

There are seven new executable statements: `CHANNEL`, `MERGER`, `MOVEPORT`, `SEND`, `RECEIVE`, `ENDCHANNEL`, and `PROBE`. Each of these takes as arguments a list of control specifiers, termed a *control information list*. The `SEND` and `RECEIVE` statements also take other arguments. A control information list can include at most one of each specifier, except those that name ports. The number of allowable port specifiers varies from one statement to another. The first three of these statements are as follows.

channel_statement	::	`CHANNEL([`*channel_control*`]`$^{(1)}$`)`
merge_statement	::	`MERGER([`*merge_control*`]`$^{(1)}$`)`
moveport_statement	::	`MOVEPORT([`*moveport_control*`]`$^{(1)}$`)`

channel_control	::	*outport_name* | `OUT=`*outport_name* |
		inport_name | `IN=`*inport_name* |
		`IOSTAT=`*storage_location* | `ERR=label`
merge_control	::	*outport_specifier* | `OUT=`*outport_specifier* |
		inport_name | `IN=`*inport_name* |

		IOSTAT=*storage_location* | ERR=label
moveport_control	::	*port_name* | FROM=*port_name* |
		port_name | TO=*port_name* |
		IOSTAT=*storage_location* | ERR=label
outport_specifier	::	*outport_name* | data_implied_do_list
outport_name	::	*port_name*
inport_name	::	*port_name*
port_name	::	variable_name | array_element_name

A CHANNEL statement must include two port specifiers, and these must name an out-port and an in-port of the same type. If the strings OUT= and IN= are omitted, these specifiers must occur as the first and second arguments, respectively.

A MERGER statement must include at least two port specifiers, and these must name an in-port and one or more unique out-ports, all of the same type. If the strings OUT= and IN= are omitted, the out-port specifiers must precede the in-port specifier, which must precede any other specifiers,

In a MOVEPORT statement, the port specifiers must name two in-ports or two out-ports, both of the same type. If the strings FROM= and TO= are omitted, these specifiers must occur as the first and second arguments, respectively. The first then specifies the "from" port and the second the "to" port.

The other four statements are as follows.

send_statement	::	SEND([*send_control*]$^{(1)}$) [*argument*]
receive_statement	::	RECEIVE([*recv_control*]$^{(1)}$) [*variable*]
close_statement	::	ENDCHANNEL([*send_control*]$^{(1)}$)
probe_statement	::	PROBE([*probe_control*]$^{(1)}$)
send_control	::	*outport_name* | PORT=*outport_name* |
		IOSTAT=*storage_location* | ERR=label
recv_control	::	*inport_name* | PORT=*inport_name* |
		IOSTAT=*storage_location* | ERR=label | END=label
probe_control	::	*inport_name* | PORT=*inport_name* |
		ERR=label | IOSTAT=*storage_location* | EMPTY=*storage_location*
storage_location	::	variable_name | array_element_name
argument	::	expression |
variable	::	variable_name | array_element_name | array_name

If a port specifier does not include the optional characters PORT=, it must be the first item in the control information list. A *storage_location* specified in an IOSTAT= or EMPTY= specifier must have integer and logical type, respectively.

2.4 Mapping

The mapping annotations LOCATION and SUBMACHINE are appended to process calls:

process_call LOCATION(*indices*)
process_call SUBMACHINE(*indices*)

where *indices* has the same syntax as the arguments to an array_element_name.

2.5 Restrictions

Port variables cannot be named in EQUIVALENCE statements. Programs cannot include COMMON data; PROCESS COMMON must be used instead.

3 Concurrency

With two exceptions, a process executes sequentially, in the same manner as a Fortran program. That is, each statement terminates execution before the next is executed. The two exceptions are the process block and the process do-loop, in which statements execute *concurrently*. That is, the processes created to execute these statements may execute in any order or in parallel, subject to the constraint that any process that is not blocked (because of a RECEIVE applied to an empty channel) must eventually execute. A process block or process do-loop terminates, allowing execution to proceed to the next statement, when all its process and subroutine calls terminate.

A process can access its own process common data but not that of other processes. By default, process arguments are passed by value and copied back to the parent process, in textual and do-loop iteration order, upon termination of the process block or process do-loop in which the process is called, or upon termination of the process, if the process does not occur in a process block or do-loop. A dummy argument declared INTENT(INOUT) is treated in the same way. If a dummy argument is declared INTENT(IN), then the corresponding parent argument is not updated upon termination. If a dummy argument is declared INTENT(OUT), the value of the variable is defined to a default value upon entry to the process.

4 Channels

Processes communicate and synchronize by sending and receiving values on typed communication streams called *channels*. A channel is created by a CHANNEL statement, which also defines the supplied in-port and out-port to be references to the new channel. A channel is a first-in/first-out message queue. An element is appended to this queue by applying the SEND statement to the out-port that references the channel. This statement is asynchronous: it returns immediately. An element is removed from the queue by applying the RECEIVE statement to the in-port that references the channel. This statement is synchronous: it blocks until a value is available. The ENDCHANNEL statement appends an end-of-channel (EOC) message to the queue. The `MOVEPORT` statement copies a channel reference from one port variable to another.

These statements all take as arguments a control information list (*cilist*). The optional `IOSTAT=`, `END=`, and `ERR=` specifiers have the same meaning as the equivalent Fortran I/O specifiers, with end-of-channel treated as end-of-file, and an operation on an undefined port treated as erroneous. An implementation should also provide, as a debugging aid, the option of signaling an error if a `SEND`, `ENDCHANNEL`, or `RECEIVE` statement is applied to a port that is the only reference to a channel.

`SEND(`*cilist*`)` $\mathtt{E}_1,\ldots,\mathtt{E}_n$ Add the values $\mathtt{E}_1$, ..., $\mathtt{E}_n$ (the sources) to the channel referenced by the out-port named in *cilist* (the target). The source values must match the data types specified in the port declaration, in number and type.

`RECEIVE(`*cilist*`)` $\mathtt{V}_1,\ldots,\mathtt{V}_n$ Block until the channel referenced by the in-port named in *cilist* (the target) is nonempty. If the next value in the channel is not EOC, move values from the channel into the variables $\mathtt{V}_1$, ..., $\mathtt{V}_n$ (the destinations). The destination variables must match the data types specified in the port declaration, in number and type.

`ENDCHANNEL(`*cilist*`)` Append an EOC message to the channel referenced by the out-port named in *cilist*.

`MOVEPORT(`*cilist*`)` Copy the value of the port specified "from" in *cilist* (the source) to the port specified "to" (the target), and set the source port to undefined.

A port is initially *undefined*. An undefined port becomes defined if it is included in a `CHANNEL` (or `MERGER`: see below) statement, if it occurs as a destination in a `RECEIVE`, or if it is named as the target of a `MOVEPORT` statement whose source is a defined port. Any other statement involving an undefined port is erroneous.

Application of the `ENDCHANNEL` statement to an out-port causes that port to become undefined. The corresponding in-port remains defined until the EOC message is received by a `RECEIVE` statement, and then becomes undefined. Both in-ports and out-ports become undefined if they are named as the source of a `SEND` or `MOVEPORT` operation.

Storage allocated for a channel is reclaimed when both (a) either the out-port has been closed, or the out-port goes out of scope or is redefined, and (b) either EOC is received on the in-port, or the in-port goes out of scope or is redefined.

5 Nondeterminism

The `MERGER` and `PROBE` statements are used to specify nondeterministic computations. `MERGER` is identical to `CHANNEL`, except that it can define multiple out-ports to be references to its message queue. Messages are added to the queue as they are sent on out-ports, with the order of messages from each out-port being preserved and all messages eventually appearing in the queue. An EOC value is added to the queue only after it has been sent on all out-ports.

The `PROBE` statement is used to obtain status information for a channel. It can be applied only to an in-port. The `IOSTAT=` and `ERR=` specifiers in its control list are as in the Fortran `INQUIRE` statement. A logical variable named in an `EMPTY=` specifier is assigned the value true if the channel is known to be empty, and false otherwise. Knowledge about sends is presumed to take a non-zero but finite time to become known to a process probing an in-port. Hence, a `PROBE` of an in-port that references a nonempty channel may signal true if the channel values were only recently communicated. However, if applied repeatedly without intervening receives, `PROBE` will eventually signal false, and will then continue to do so.

6 Mapping

The `PROCESSORS` declaration and the `LOCATION` and `SUBMACHINE` annotations have no semantic content, but determine performance by specifying how processes are to be mapped within an N-dimensional array of processors ($N \geq 1$).

The `PROCESSORS` declaration is analogous to a `DIMENSION` statement: it declares the shape and dimensions of the processor array that is to apply in the program, process, or subroutine in which it appears. As we descend a call tree, the shape of this array can change, but its size can only become smaller, not larger.

A `LOCATION` annotation is analogous to an array reference. It specifies the virtual processor on which the annotated process is to execute. The specified location cannot be outside the bounds of the processor array specified by the `PROCESSORS` declaration.

The `SUBMACHINE` annotation is analogous to an array reference in a subroutine call. It specifies that the annotated process is to execute in a virtual computer with its first processor specified by the annotation, and with additional processors selected in array element order. These processors cannot be outside the bounds of the processor array specified by the `PROCESSORS` declaration.

References

[1] Chandy, K. M., and Foster, I., A deterministic notation for cooperating processes, Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993.

[2] Foster, I., and Chandy, K. M., Fortran M: A language for modular parallel programming, Preprint MCS-P327-0992, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

[3] Foster, I., Olson, R., and Tuecke, S., Programming in Fortran M, Technical Report ANL-93/26, Argonne National Laboratory, Argonne, Ill., 1993.

[4] *Programming Language Fortran*, American National Standard X3.9-1978, American National Standards Institute, 1978.